15/16

NOTE DI SOFTWARE

ARCHIVIO STORICO
ASSOCIAZIONE POZZO DI MIELE
RPDM-116
RNSW-112

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA

Honeywell
Honeywell Information Systems Italia

Note di software.
è un bollettino trimestrale, edito a cura del Centro di Ricerca e Progettazione della Honeywell Information Systems Italia e dell'Istituto di Cibernetica dell'Università di Milano.

Note di software.
intende costituire uno strumento per la rapida e informale circolazione di informazioni, idee ed esperienze nell'area della tecnologia del software. In particolare, intende stimolare il dibattito sulle metodologie orientate alla diminuzione del rapporto costo/qualità dei programmi, e sugli strumenti atti ad automatizzare l'attività di produzione del software in ogni suo aspetto.

La collaborazione a **Note di software** è libera.
In particolare, si sollecitano contributi nella forma di brevi monografie o bibliografie essenziali commentate sui seguenti temi: tecniche e strumenti per la programmazione strutturata e modulare, metodologie e strumenti per il testing e il debugging dei programmi, documentazione del software, aspetti organizzativi e gestionali nella costruzione di sistemi software di grandi dimensioni.

Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Comitato di Redazione (HISI - Via ai Laboratori Olivetti - Pregnana Milanese), o di inviare direttamente il dattiloscritto in triplice copia.
I contributi non dovrebbero superare le quattromila parole.

I lavori accettati per la pubblicazione non vengono revisionati da un comitato tecnico. Pertanto, ogni contributo pubblicato riflette le opinioni personali dei suoi autori, e non necessariamente quelle del Comitato di Redazione.
Eventuali revisioni saranno effettuate di comune accordo fra il Comitato di Redazione e i proponenti.

Note di software.
viene distribuito a chi ne faccia richiesta alla redazione.

Direttore responsabile: Paolo Ghelfi
Comitato di Redazione: Roberto Polillo - Wanda Anselmetti
Redazione: Franco Soresini

15/16

Luglio 1981

FPDM-116

NOTE DI SOFTWARE

Indice:

QUESTO NUMERO .... pag. 1

DISCUSSIONI:

— HOW, WHAT, WHY, di G. Degli Antoni, M. Maiocchi, R. Polillo, B. Zonta ” 2

CONTRIBUTI TECNICI:

— A DOCUMENTATION METHOD AND THE RELATED TOOL FOR A SOFTWARE ENGINEERING ENVIRONMENT, di A. Chiapparino, A. Cicu, M. Maiocchi ” 8

— NOTE SULL'USO DI FUNZIONALITÀ COBOL IN AMBITO PHOS, di C. Magnoni, E. Spoletini, E. Toscani ” 22

— PROGETTAZIONE E REALIZZAZIONE DI APPLICAZIONI IN AMBIENTE TP, di T. Bettinazzi, A. Maserati, F. Monzani, E. Spoletini, E. Toscani ” 37

— THE RIGHT WAY TO COMMUNICATE ACROSS DISTRIBUTED MACHINES, di P. Grandi ” 46

— NOTE SU UNA TECNICA DI DOCUMENTAZIONE DI PROGETTO, di E. Bertolotti ” 60

IL SEGNALIBRO ” 64

Autorizzazione del Tribunale di Milano n. 87 del 23 febbraio 1977

QUESTO NUMERO...

La specificazione dei requisiti di un sistema software, la documentazione di progetto, la metodologia PHOS e i meccanismi di comunicazione nell'ambito di sistemi distribuiti sono i temi trattati in questo numero doppio di NOTE DI SOFTWARE.

Nel primo articolo, vengono discussi i rapporti fra le esigenze che stanno alla base di un sistema software (il "perché"), le funzionalità da esso fornite (il "che cosa") e la sua realizzazione (il "come"). In tale discussione, G. Degli Antoni, M. Maiocchi, R. Polillo e B. Zonta propongono l'adozione delle reti di Petri quale strumento per modellare, in termini complessivi, il sistema (descrizione del "comportamento desiderato"), e di concetti mutuati dalla teoria del controllo per "controllare" il comportamento reale del sistema sulla base del comportamento desiderato.

Nel secondo contributo, A. Chiapparino, A. Cicu e M. Maiocchi, dopo una analisi critica dei problemi che spesso si incontrano nell'area della documentazione del software, descrivono il metodo DOC (Documentation on Context), e lo strumento che lo supporta (Documentor), per la generazione, riproduzione e manutenzione della documentazione tecnica in un ambiente di produzione di software.

La metodologia PHOS permette di progettare la struttura di un programma "deducendola" dalla struttura dei suoi dati di ingresso e uscita. Nel terzo lavoro, C. Magnoni, E. Spoletini ed E. Toscani mostrano, con un esempio, come tale metodologia possa essere adottata con profitto anche quando il linguaggio di programmazione utilizzato per la codifica metta a disposizione del programmatore funzionalità sofisticate (nell'esempio, il Cobol con funzionalità di "sort interno" e di "report writer").

Nel quarto lavoro, T. Bettinazzi, A. Maserati, F. Monzani, E. Spoletini ed E. Toscani indicano come le linee guida di PHOS, opportunamente adattate, costituiscano una valida base metodologica anche per la progettazione di applicazioni TP. La metodologia risultante, PHOS-TP, permette di costruire programmi transazionali a partire dalla struttura del dialogo operatore-sistema. La metodologia è presentata attraverso un semplice esempio.

Il quinto contributo è dedicato ai meccanismi di comunicazione fra processi nell'ambito di sistemi distribuiti. Dopo un'ampia discussione sui modelli concettuali più adatti a rappresentare la comunicazione in tali sistemi, P. Grandi propone una primitiva di comunicazione basata sul concetto di "baratto" fra due partner, e ne descrive, in un linguaggio ad alto livello, una possibile realizzazione.

Infine, nella sesta nota, E. Bertolotti descrive una esperienza di adozione, nell'ambito del progetto di un sistema hardware e software, di una tecnica di documentazione legata al metodo PSPN.

La Redazione

DISCUSSIONI

HOW, WHAT, WHY

G. Degli Antoni(°), M. Maiocchi(°°), R. Polillo(°°), B. Zonta(°°°)

INTRODUCTION

It seems that the various efforts that have been undertaken in order to limit the consequences of the so-called software crisis have not yet reached definitely convincing results. Many approaches have been followed:

- new conceptual tools have been suggested (data structures, exceptions, ...)
- programming languages have embodied these suggestions
- tools to automate certain aspects of software production have been introduced
- methodologies to increase the discipline in various steps of the production processes have been suggested.

Despite this extremely wide effort, a clear picture of the entire field has not been given.
In this situation, one has the impression that the efforts made lack criticism and it seems very difficult to evaluate the existing proposals and even to classify them.
Emerging conclusions can neverthless be traced:

- structured programming is more and more used in teaching programming
- attention is directed to the entire software life-cycle, particulary to the phases that precede programming
- bootstrapping has shown the real possibility of increasing the productivity, the quality and the portability of large pieces of software (compilers).

(°) Institute of Cybernetics University of Milano

(°°) Institute of Cybernetics University of Milano and Honeywell I.S.I

(°°°) C.N.R., Milano

The foregoing conclusions are encouraging enough to take new steps ahead in order to clarify the entire field.

One step has been recently more or less directly indicated by T. Winograd: to integrate most of the results from the many disciplines of programming practice and research in a single general purpose tool.

It is quite difficult to evaluate such a proposal now, and we will avoid any criticism. We want only to agree with that proposal when it suggests that the entire complexity of the problem, as actually results from practice, has to be taken into account.
In particular, one has to take into account that software problems practically never start from scratch: there exists in all cases something that has been developed before. Moreover, software systems are always connected with some unpredictable behaviour, due to incorrect human intervention or to malfunctions in hardware or in software.

With all the indicated premises, the present note does not claim to contribute another new methodology or technology. The unique claim is that a conceptual view has to be taken if one has to contribute suggestions to the field. We will give a few of them after presenting a coincise point of view of the entire field.

COMPUTER PRACTICE AND CONTROL

All actual computer problems are control problems.
At the simplest extreme level, a piece of code has to control the behaviour of a machine. At this extreme, all the views are idealized: the code is considered an abstract device and machine mistakes are fully ignored. At the highest level, the situation is definitely more complex and difficult.

In fact, at the highest level there are machines (possibly unreliable) working in an environment where many partially independent entities work frequently in an unpredictable way. Despite this complex situation some definite goal of control has to be obtained. To master such a situation seems to be (and perhaps it is) a tremendous task. Nevertheless, something has to be said.

To have observed that there exists some control objective is certainly not fully trivial. Nevertheless, it is necessary to detail the overall picture carefully if one hopes to be able to use such an observation in some sense. In particular, one would hope that understanding control problems would introduce feedback ideas in software practice. Hopefully one will try to take advantages from introducing feedback ideas similar to those that came out from the control theory.

Trying to think of the computer use as a control problem, one has to introduce a control terminology. For example, at the simple extreme of control technology one needs to know if some control actions are taken on the basis of an actual behaviour as compared with some "reference" (fig. 1)
If one will be able to sketch down as in the mentioned figure, then some ideas from the control discipline will be useable.

Obviously, we are facing a new problem, because the control discipline and feedback ideas are mostly useless for problems similar to those mastered in computer science. Nevertheless, we will try this approach. The reader who wants to come to a conclusion soon is invited to think for example of the reason for the existence of operating people in computer rooms. If the example is too difficult (and it is), let's remember that frequently happens to programmers that vacation interruptions are required in order to "put in order" a situation in which something unpredictable has occurred in connection with the use of a set of programs (procedures) not necessarily incorrect. In this case the programmer is requested to take action (actually a control action) in order to achieve what is requested by the procedure.

The example clarifies many aspects of computer practice:

- the existence somewhere of a "desired behavior" or reference behavior
- the need for human intervention in some unpredicted (not unpredictable) situation
- the human intervention as "control action".

Yet at this stage of detail many conclusions can be drawn: the most important being the need for understanding fully the limitation of automating processes in view of control actions, particularly in cases where damages could result.

CONTROL AND MODELLING

The discussion in the preceding section has shown that, in actual control systems, control actions are not fully based on predetermined rules.
Obviously, the reasons for such a situation could be many.
With a control terminology we could have:

- an actual situation different from the modelled one
- a lack of control action in the automated part of the procedure.

In both cases it is clear that the "desired behavior" or "reference behavior" is mandatory both for the automated control actions and for the human intervention.
It is clear, moreover, that the "desired behavior" (we will use such a term from now on) requires a modelling of the system to be controlled.

To clarify the last sentence a little, let us return now to fig. 1. In the case in which control actions are fully effective, then the actual behavior will correspond exactly to the desired one.
In a more complicated language, as suggested by a theorem due to R.C. Conant and W.R.Ashby, "every good regulator of a system must be a model of that system".
We are now in a definitely better situation: in fact, if we know the desired behavior, we know the meaning of control actions: to force the system to behave as closely as possible to the desired one.
But we now know more: that the building up of the desired behavior requires careful modelling of the system to be controlled.

There are still many implications in the above discussion:

- the fiction of full control suggest that the behavior of the system under control action can be described by the desired behavior

- the desired behavior does not need to detail the control actions to be taken in order to reach the control goal.

This last conclusion is important, and deserves some discussion. In fact, being able to introduce the desired behavior corresponds to introducing a conceptual component of the system under examination. Another component will be the regulator sending control actions to the controlled system. Such a decomposition corresponds to a structuring discipline that, hopefully, will simplify the design.

As a conclusion for the present section, let us note that the introduction of the desired behavior could have the following distinct benefits:

- the desired behavior is a descriptive model of the actual behavior
- the desired behavior is the source for the control actions
- the desired behavior is the basis for the design.

The whole discussion seems to suggest the need for selecting a formal description of the desired behavior as one of the basic components of automated control systems.

THE FORMAL DESCRIPTION OF THE DESIRED BEHAVIOR

To formalize the desired behavior requires suitable conceptual tools. In fact, if one has to model systems such as those mentioned before, in which computers and humans interact, one has to model systems where causal aspects have to be taken into account and where concurrency (partial independence) and conflicts (non determinism due to partial knowledge) occur.

The constraints of the description are, moreover, that such a description has to serve many different purposes as indicated in the preceding chapter. Every purpose is related to a class of users. As a consequence, the desired behavior needs to be understandable by:

- the users of the system
- the designers of the systems
- .....

Such a wide scope of users of the desired behavior stresses the need for a very careful choice of the level of the language.

The authors strongly suggest, for all the indicated reasons, that that conceptual frame in which the desired behavior of actual systems would be represented with the desired level of abstraction has to be built in the frame of the General Net Theory as developed by Petri, Holt and others.

We will avoid details about this subject, referring the reader to the literature. We will only underline the main reasons for such a choice, besides those already indicated.

In the General Net Theory, starting from an extremely abstract notion of condition-event net, one can build through morphisms higher levels of description corresponding to the requested ones. Morphisms are mappings similar (but more powerful and more general) to those obtained with the top-down development. The availability of those morphisms is not only a design approach, but permits the development of rigor in the definition of higher level nets starting from the basic condition-event nets.

The advantage of using General Net Theory can be appreciated if one considers that the meanings of a net can be given in terms of other nets and morphisms that map the latter into the former. With such a view, the desired behavior does not give details but it well represents abstractly more detailed behaviors.

To give a pictorial image of the use of the desired behavior, the reader can imagine a large net (at the level of abstraction corresponding to the user needs) for example of the type described in [3], in which marks (corresponding to structured pieces of information) give the image of the actual situation with respect to the system status.

The positions of the marks on the net suggest actions on the controlled system and the advancement of the positions of the marks. Briefly stated, the reference processes are generated by a net in which conditions hold as a consequence of actions from the system to be controlled and as consequence of control actions.

FROM THE DESIRED BEHAVIOR TO THE REGULATOR

The desired behavior describes concisely desired trajectories (with possibly concurrent actions). The actual trajectories are selected by the past behavior and could depend on the future one.

The lock of the actual behavior to the desired one depends on the effectiveness of the control actions taken by the regulator on the system to be controlled (fig. 1).

To build up the regulator is obviously the main task of the software as stated in these notes. In fact, the desired behavior for a large class of problems is based on traditions well developed in electronic data processing practice. Designing a regulator has a simple theoretical interpretation: to build up a larger net representing all possible behaviors and a morphims transforming the new net into the desired behavior.

The observation is by itself interesting enough, because it permits the development of technical control terminology in the frame of General Net Theory. Nevertheless, it is too general and does not give any suggestion for the design of the regulator.

In order that control be effective, one has to establish communications links between the regulator and the system to be controlled. Through these links many kinds of signal will flow. It is from the nature of these signals that we classify the nature of the regulation processes:

a) Signals correspond to the presence of marks (condition holding) in the places of the desired behavior, and flow from the regulator to the system to be controlled. The control scheme is very simple: signals trigger external actions (in the controlled system) and the events of the desired behavior occur correspondingly. The effectiveness of the signal on the system behavior is taken by confidence.

b) Acknowledgment signals flow from the system to the regulator. In this case the next steps (possibly concurrent) in the building up of the desired behavior will be controlled by acknowledgments and by the conditions that have determined the considered actions.

c) If some action could not take place in the system to be controlled then some kind of signal has to flow towards the regulator.

It is important to note that this last point corresponds to a suggestion due to Holt and extended by Genrich, Lautenbach and Thiagaragiar in order to build up structured nets with some kind of well behaved behavior.

The control schemes resulting from the first two approaches could be called "case dependent" control schemes, as they are the consequence of the definition of "case" in Petri net terminology. In such a terminology, a case is just a marking of a net. In the first control scheme, the "cases" of the desired behavior determine the control actions. In the second scheme, the answers correspond to the "case" of the system to be controlled as indicated by acknowledgments, with the confidence that these correspond to an actuated action following the desired one. The third scheme corresponds to taking into account a different kind of marks. The actions to be taken will be actually subject matter dependent and can not be discussed in general terms.

Obviously, not all schemes are general enough. Moreover, various difficulties have been neglected. The main one is due to the problem of establishing the concurrence of conditions holding. In fact, if one is waiting for a new case (a set of holding conditions) and actually only some of them are observed, then some action has to be taken:

- to wait under the hypothesis that the measurement has been made too early but with the confidence that all conditions will hold

- to wait just until some time-out end. If the situation is not modified in the desired direction, then some action has to be taken.
  The action still is subject matter dependent.

The above discussion is not the only one that can be suggested by the proposed picture. In fact, more sophisticated control schemes can be derived in general terms without reference to the subject matter to which the regulator is directed (process control, edp, operating systems,...).

To put the feasibility in general terms, it is enough to consider that, as a response to the desired behavior, one can measure some syndrome suitable to establish, for example, if some data is noisy or for some reason is subjected to errors.

Then by evaluation of the syndrome, various approaches can be followed:

- retry operations

- advancing the desired behavior in the direction in which the missing data is not needed while concurrently retrying in order to request a correction.

This last control scheme needs to embody some degree of intelligence and the capability of analyzing the desired behavior.

All suggested control schemes can be implemented in a large variety of ways:

a) control actions are left to a human operator

b) control actions are directly left to programs that implement the selected control scheme

c) control actions are left to a computer with some degree of intelligence, that implements the capability of selecting control actions in a large variety of situations but is able, for example, to accept a different desired behavior as a parameter.

It is extremely important to note that, despite all our efforts, a human operator is mandatory. The reason for this situation begins to be clearly understood as the consequence, for example, of the cuncurrency problem indicated.

It is interesting to note that different approaches, with the same scope, seem to be implied in Petri paper devoted to "modelling as a communication discipline" and in the recent lecture notes of Holt devoted to "representing processes".

TAXONOMY

One of the purposes of the suggested picture is to introduce structure in the design of complex software systems.

The components of the structure will be used in methodologies by which pieces of code will be finally produced. In order to contribute more to the understanding of the entire process of software production, it could be of some interest both to connect the proposed scheme with the actual terminology of the software life-cycle, extending the well known computer science terminology in which a clear distinction is made between *syntax* and *semantics*.

So we will agree that *syntax* corresponds to the programs that will actually be succinctly described by a name in the net describing the desired behavior as events.
For example, a name could be "compile the document". Such a command corresponds to a piece of code, just a part of the *syntax*. Obviously such a code has some functionality, or equivalently it computes something as defined by specifications, if one assumes the correctness of the considered program. Specifications actually correspond to the realm of the *semantics* of the considered programs.

But *why* such a program (*how*) performs such a computation (*what*)?

The answer can be given if one looks at the level of abstraction selected by the desired behavior. If one takes such a desired behavior as the motivation for the choice of the specifications of the programs, then the term *pragmatics* seems to be justified.

But there are many other reasons to select the term pragmatics in order to indicate the desired behavior.

Two of them are:

- in linguistics, the term pragmatics seems to indicate something related to the *use* of semantical functions in a communication frame, similar to what happens in the proposed approach, where the desired behavior interconnects various functions in a common purpose (i.e. the desired behavior).

- Petri has suggested the development of a "formal pragmatics" as a discipline in which a formal attitude is selected in the description of systems and organizations.

The above taxonomy does not enlighten all the aspects of the implementation. In particular, it is to be noted that, following the control schemes, there will be more or less complex pieces of software devoted to the management of the actions to be taken.

Such a structural part of the considered software is outside the suggested taxonomy, if it is not considered just as the implementation of the pragmatical aspects of the system. For what concerns the relationship with the software life cycle, it is difficult to take a single clear picture.
Nevertheless, it is not impossible to recognize in the requirement analysis the process driving to some piece of documentation that could actually be used as the desired behavior.

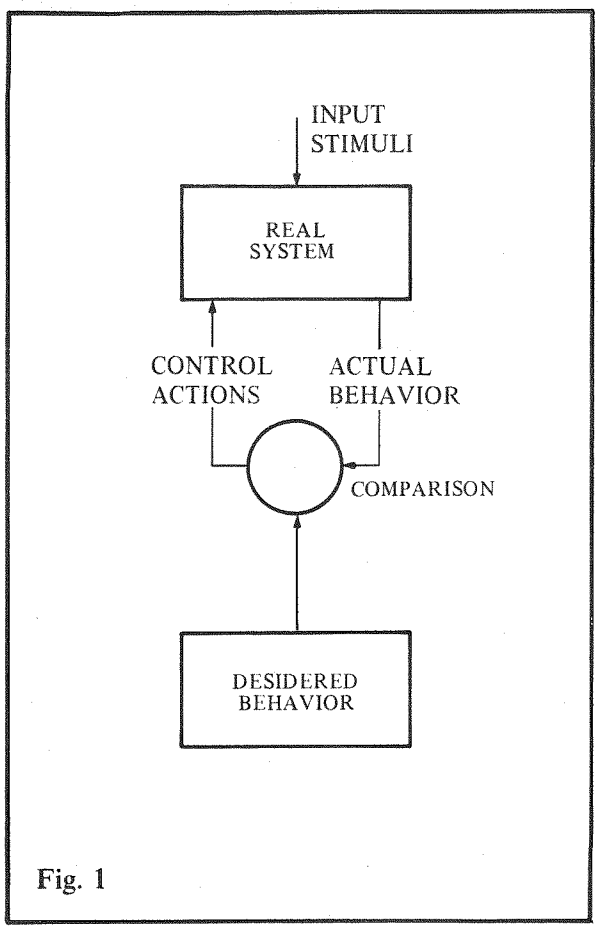


Fig. 1

BIBLIOGRAPHICAL NOTE

[1] W.R Ashby, R.C. Conant, EVERY GOOD REGULATOR OF A SYSTEM MUST BE A MODEL OF THAT SYSTEM, Systems Science, 1,2, 1970

[2] V. De Antonellis, G. Degli Antoni, G. Mauri, B. Zonta, EXTENDING THE ENTITY-RELATIONSHIP APPROACH TO TAKE INTO ACCOUNT HISTORICAL ASPECTS OF SYSTEMS, in Proc. of Intern. Conf. on Entity-Relationship Approach to System Analysis and Design, Los Angeles, 8-12 Dec. 1979

[3] G. Castelli, G. Haus, M. Maiocchi, A METHODOLOGY FOR THE CONSTRUCTION OF FUNCTIONAL SPECIFICATIONS FOR SOFTWARE PROCEDURES AND PROGRAMS, Third International Honeywell Conference, Minneapolis, 1979

[4] G. Degli Antoni, B. Zonta, DOPO LA PROGRAMMAZIONE STRUTTURATA, convegno AICA, 1977, Pisa

[5] H. J. Genrich. P.S. Thiagarajan, BIPOLAR SYNCHRONIZATION SYSTEMS, Advanced Course on General Net Theory of Processes and Systems, Lectures Notes in Computer Science, Springer Verlag, 1980

[6] A.W. Holt, PROCESS REPRESENTATION, Notes of S. Cruz Seminar on Programming Methodology, 1979

[7] C.A. Petri, MODELLING AS A COMMUNICATION DISCIPLINE, Measuring, Modelling and Evaluating Computer Systems, North Holland Publ., 1977

[8] C.A. Petri, GENERAL NET THEORY, Advanced Course on General Net Theory of Processes and Systems, Lecture Notes in Computer Science, Springer Verlag, 1980

[9] T. Winograd, BEYOND PROGRAMMING LANGUAGES, Comm. ACM, July 1979

CONTRIBUTI TECNICI

A DOCUMENTATION METHOD AND THE RELATED TOOL FOR A SOFTWARE ENGINEERING ENVIRONMENT

A. Chiapparino(°), A. Cicu(+), M. Maiocchi(*)

Riassunto

Questo articolo descrive un semplice metodo (il metodo DOC, acronimo per 'Documentation On Context') e il relativo strumento (Documentator), utilizzabili per la generazione, riproduzione e manutenzione di documentazione tecnica in un ambiente di produzione di software.
Il metodo si integra con lo standard di autodocumentazione dei programmi adottato nei laboratori di Pregnana della Honeywell I.S.I., ed è ora ampiamente diffuso in tale ambiente.

Abstract

The paper illustrates a simple method (the DOC method, acronym for 'Documentation on Context') and the related simple tool (Documentator) used for generation, reproduction and maintenance of documentation for a software production environment. The method is integrated with the program self-documentation standard used in HISItalia Software Engineering Laboratories of Pregnana Milanese and is now in large diffusion in this environment.

1. INTRODUCTION: THE IMPORTANCE OF DOCUMENTATION IN SOFTWARE DEVELOPMENT

The importance of software documentation in a production environment is well recognized.

Software is an industrial product by its nature intrinsically visible and usable only if accompanied by clear and complete documentation: documentation, being the unique visible entity in software production, becomes the fundamental medium for human communication ((1) and fig. 1).

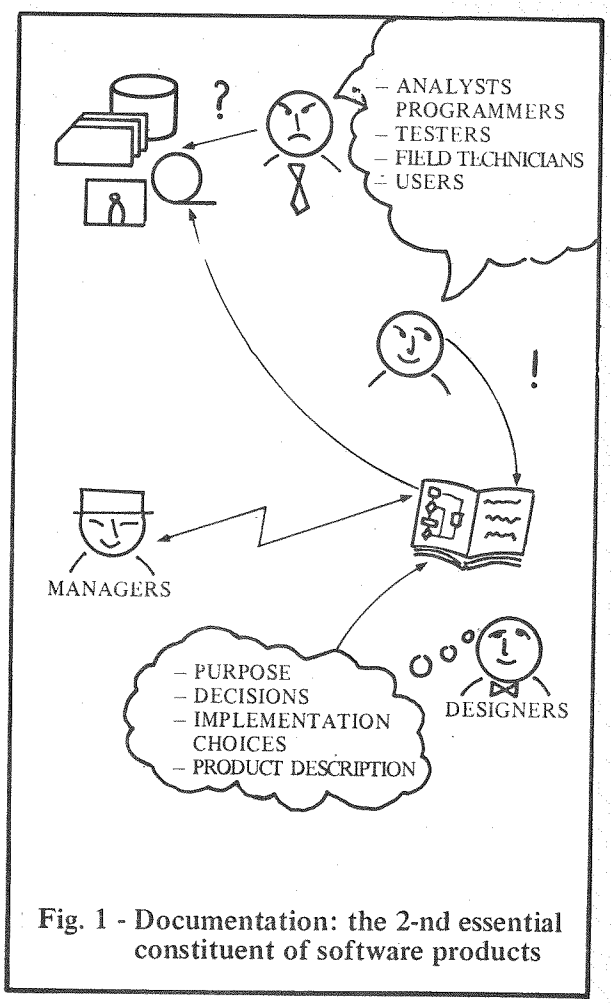


Fig. 1 - Documentation: the 2-nd essential constituent of software products

(°) Etnoteam srl, Milano

(+) Honeywell Information Systems Italia, Pregnana Milanese

(*) Università di Milano, Istituto di Cibernetica

The present work has been done in the frame of the Communication and Programming Project, joint research project between HISItalia and University of Milano.

The present paper has been presented at the AICA Conference, Bologna, 1980.

Documentation is the basis of all software life-cycle phases:

— conception
— functional definition
— design
— coordination of the implementation
— testing
— evaluation of quality
— installation
— usage
— maintenance.

Ease of use depends heavily on documentation.

We can conclude: in addition to the code, documentation is the other essential constituent of a software product.

From the management point of view, documentation is a kind of map for work organization, allowing auditing activities and work progress control.

Before examining the software documentation method being used in HISItalia Software Engineering, it is useful to discuss the problems which affect this area of software development today, together with the possible solutions.

2. SURVEY OF PROBLEMS AND RELATED SOLUTIONS IN SOFTWARE DOCUMENTATION

2.1. Problem: Software documentation is often incomplete

Origin : • poor or missing standards/guidelines which define and make clear the contents of the documents

• insufficient push by software management in promoting and controlling the accuracy of documentation. Consequence: insufficient internal review activity which could act as strong encouragement in documenting well.

Solution: A *management* action in *pushing* and *promoting* professionality in the documentation of any project; proper *periodical reviews* are necessary to maintain the right emphasis on the control of the contents, and to confirm the importance given to documentation.

However, some prerequisites are necessary to support management and programmers:

• availability of clear and easy *standards* defining:

  • • structure of documentation

  • • content of the documents

• availability of tools (text-editors, text-formatters) which make the *updating* of documentation *easy and cheap*.

2.2. Problem: Software documentation often issued late

Origin : • lack of a standard defining:

  • • the *phases* of software development and maintenance life-cycle

  • • the *documents* to be issued at the end of each life-cycle phase.

• insufficient push by software management in getting software functions and structure specified and documented *before coding*.
In case of delays in development, the documentation activity, being undervalued, is the first one to be skipped. One aspect should also be taken into account: that, when the documentation is done after the program development, it is felt to be tedious and useless work.

• Typing of documentation (where typing has not been replaced by text-editing).

Solution: The above observations made about incompleteness, apply also here.

In this case, *standards* have to clarify

  • • the phases of software development and life-cycle

  • • which documents have to be issued at the end of each phase

Again, *text-editors* and *text-formatters* contribute to reduce time in producing documents.

But the major point is that software engineers must be made to understand that the first version of a software program is its narrative description: the code must be seen as the final detailed specification of the initial documentation. Therefore: *education on a methodology* is needed.

Finally, *each project must be planned* in terms of milestones and resources applied to each activity. As in any other field, also in software documentation is true that "an unplanned event will not happen": and the standard related to the development phases is the best support to prepare a plan.

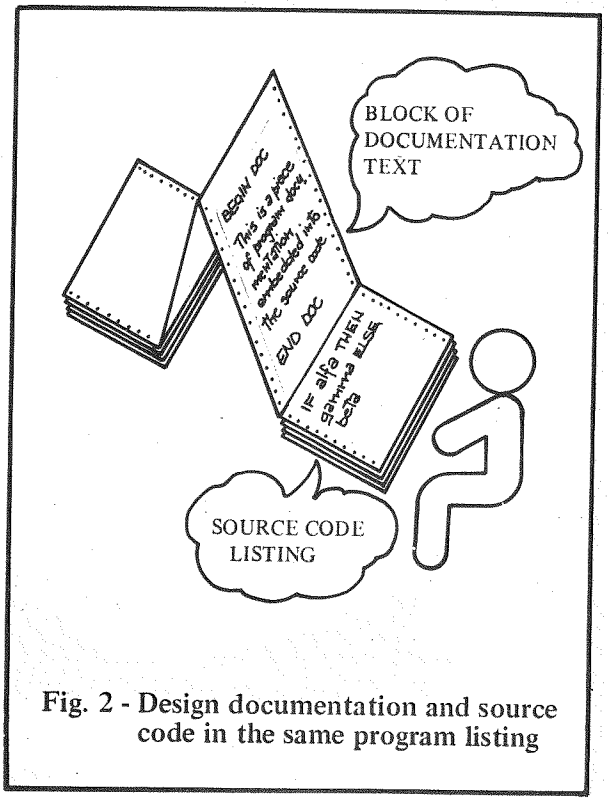


Fig. 2 - Design documentation and source code in the same program listing

2.3. Problem: Software documentation, when updated, is not updated synchronously with the programs

Origin : Software design documentation is often physically separate from the programs (this is either because design documentation is typed manually, or because, even when it is produced via text-editing, design documentation is a different document and a different file, not embedded in source program bodies).

Therefore, synchronous updating would require the programmer to update two documents at a time: this in many cases could be impractical work, and cause waste of time.

Solution: The adoption of *design documentation method* requiring design documentation of each program as a *part of the source program body*, in the form of comments.

In this way it is *natural* for the programmer to update the design documentation together with the source statement code.

When the programmer is working on the correction of a bug, or on the extension of a functionality, he has both the code and the documentation in the same program listing, so he has access to the program description two or three pages of listing before the code he is working on.

This proximity of the documentation to the related program code makes it easy for the programmer to access the program description and to update it exactly at the moment when he has his mind concentrated on the problem in the process, and not weeks later. At least the time of remembering and restudying the problem is saved.

However, with the exception of the activity of coding corrections or program extensions, it is not practical to have the documentation only in the form of comment blocks embedded in the program listing.

During the initial design phase, the project reviews, the training of new people inserted in the project, and the training of field maintenance resources, it is better to have documents without the details of the code: in this way documents are smaller in dimension, cheaper for reproduction, easier for study and consultation.

Therefore, the method of embedding documentation, in the form of comment blocks, in the program body implies the availability of a tool able to extract the comment blocks out of the source programs and to assemble them in organized documents. The proposed solution, adopted in the DOC method is presented in the next section 3.

2.4. Problem: Incongruencies are present in software documentation, due to the fact that the documentation is sometimes redundant (some concepts and descriptions are repeated in more than one document).

Origin : Redundancy is often necessary and positive.

Examples:

- user visibility aspects must be present in Engineering Functional Specifications (FS) and in user manuals
- single functionalities are described in FS, test checklists, test spec's
- architectural structure is described in FS and in Product Design Description (PDD, often used also as Software Maintenance Documentation, SMD)
- interfaces between components are described in PDD and in Integration Reports
- and so on.

The problem arises from the fact that it is difficult, for analysts/programmers, to remember in which different documents the same information is present: as a consequence, after some time, when this common information is updated, the updating is not done everywhere, and incongruencies arise (which often are the real origin of software errors).

Solution: It is the opinion of the authors that this redundancy is unavoidable, and that there are good reasons for encouraging the proper use of redundancy.

As a matter of fact, the technical description of a product is contained partly in FS (Functional Specifications, which furnish a functional description of the software product) and partly in Software Maintenance Documentation.

However, FS's contain information such as that related to performances, which is proprietary and should not be included in documentation to be distributed to customer sites.

On the other hand, some of the information contained in FS, such as the structure of the software upper levels and/or its user interfaces, are needed in Software Maintenance Documentation to provide necessary completeness of understanding by support personnel.

The above relationship between information and documents is illustrated in figure 3.

2.5. Problem: Software documentation is costly to be read because in some cases, when it is accurate, it is not concise and excessively wordy and prolix

Origin : • The difficulty of explaining concepts and structures which are often complex in a concise and clear way: this difficulty increases when the design is not structured in a top-down fashion.

• Usually no different emphasis between major and minor concepts.

Solution: A first step toward the solution of this problem is again a *set of good standards* which explain:

- the content of the document (which topics must be treated)
- the logical order of the topics and a set of suggestions which clarify how to describe a single topic.

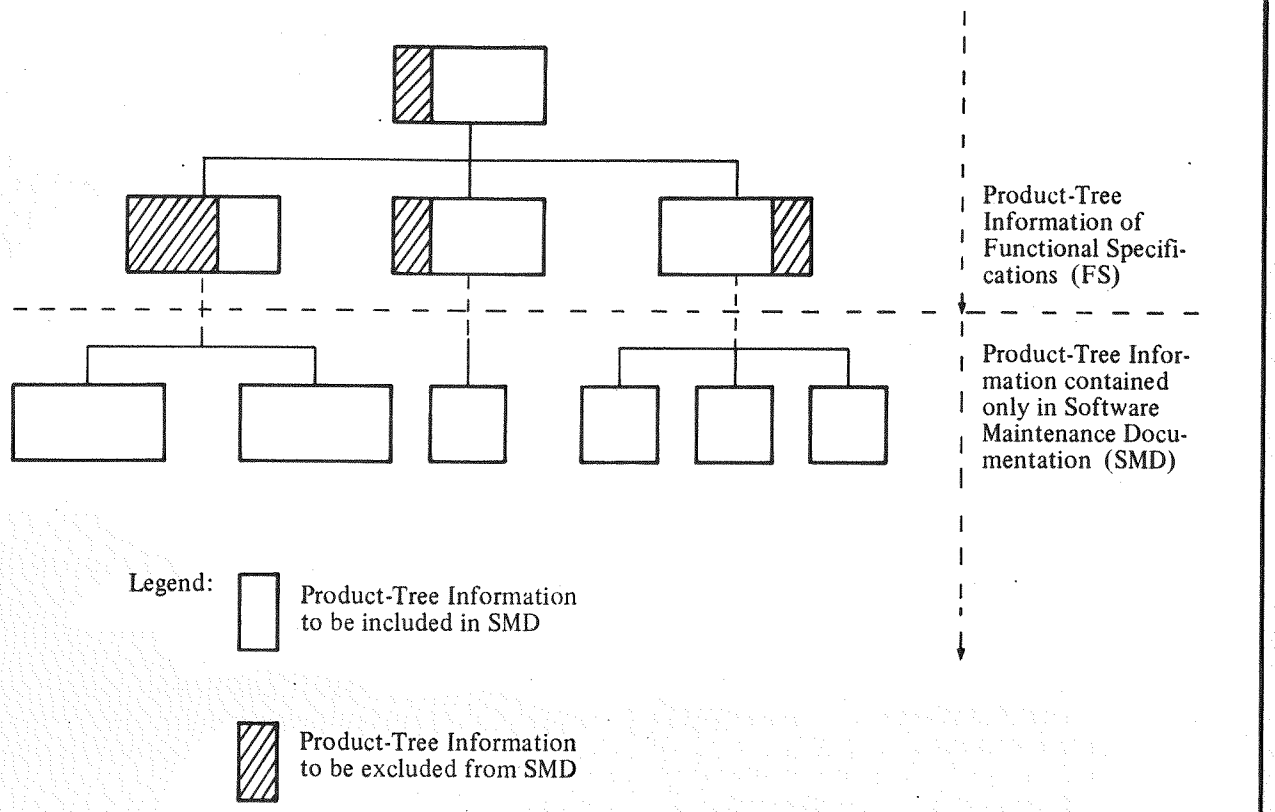


Fig. 3 - Diagram of information present in more than one document.

An example of a standard for software program self-documentation is in (3). The concepts outlined in (3) have been adopted in Honeywell Information Systems Italia Software Engineering Standard for Program Design Documentation (see Ref. (4) and (5)).
In the above standards, emphasis in put (and documentation models/forms are suggested) on the following essential points:

- data specifications
  - meaning (single values of data items included
  - relationships with program components
    - initialized by
    - set by
    - used by
- top/down structure
- program specifications
  - input
  - output
  - process
  - control from/to; calls
  - work areas

This precise guide aids the analyst/programmer in being at the same time synthetic and precise.

A second aid in producing clear and concise documentation is the usage of the PSPN method, where the main concepts are presented graphically on the left pages of the document, and the explanations on the right pages. This

method is remunerative of the effort specially for user manuals and guides, or standard/guidelines, and in production of software education courses (6), (7).

The considerations of this § 2 have shown that any approach adopted to produce a good software documentation must give great importance to standards, guidelines, planning as well as to good production methods and tools. The remaining part of the article will describe DOCUMENTOR, a tool conceived to support the above methodological aspects.

3. THE METHODOLOGICAL BASIS OF THE "DOC" METHOD AND THE FUNCTIONALITIES OF DOCUMENTOR, THE SUPPORT TOOL

DOC is a documentation method oriented to documentation of software products: as such, it does not pretend to be a generalized method of documentation, nor to be a complete method of text-editing and printout formatting.

The main objective in the design of the method was to take into account the specific nature of software products and of their documentation, and the problems which have been discussed in § 2, and to design a method for documenting software with the following characteristics:

- very easy to teach and to learn
- very cheap to be supported with an ad hoc tool DOCUMENTOR)
- open to extensions, if suggested by the experience
- very adherent to the specific aspects of software production and documentation.

3.1. Requirements for DOC method and DOCUMENTOR

A description of the specific objectives follows:

a) *Documentation on Context*

The first basic concept adopted in the design has been the following: "the right place to put software documentation texts is in the *context* of the software entities they describe" (see fig. 4).

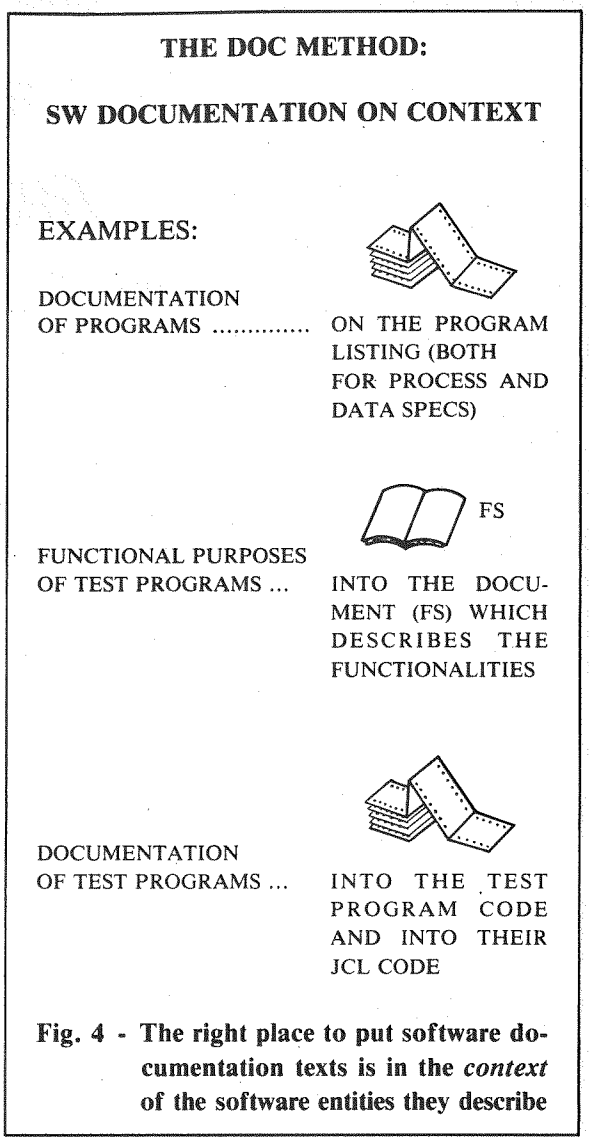


Fig. 4 - The right place to put software documentation texts is in the *context* of the software entities they describe

Reasons for above choice are intuitive as has been shown in § 2.

b) *Documentation of a software project distributed in more source components*

Usually, software products are structured in components implemented through separate source programs: it is the work of macroprocessors, compilers and linkers which assemble them in the proper way to build the finite object executable code. (See fig. 8).

One consequence is that the documentation of the product is an organized assembly of documentation blocks distributed in the different source code components.

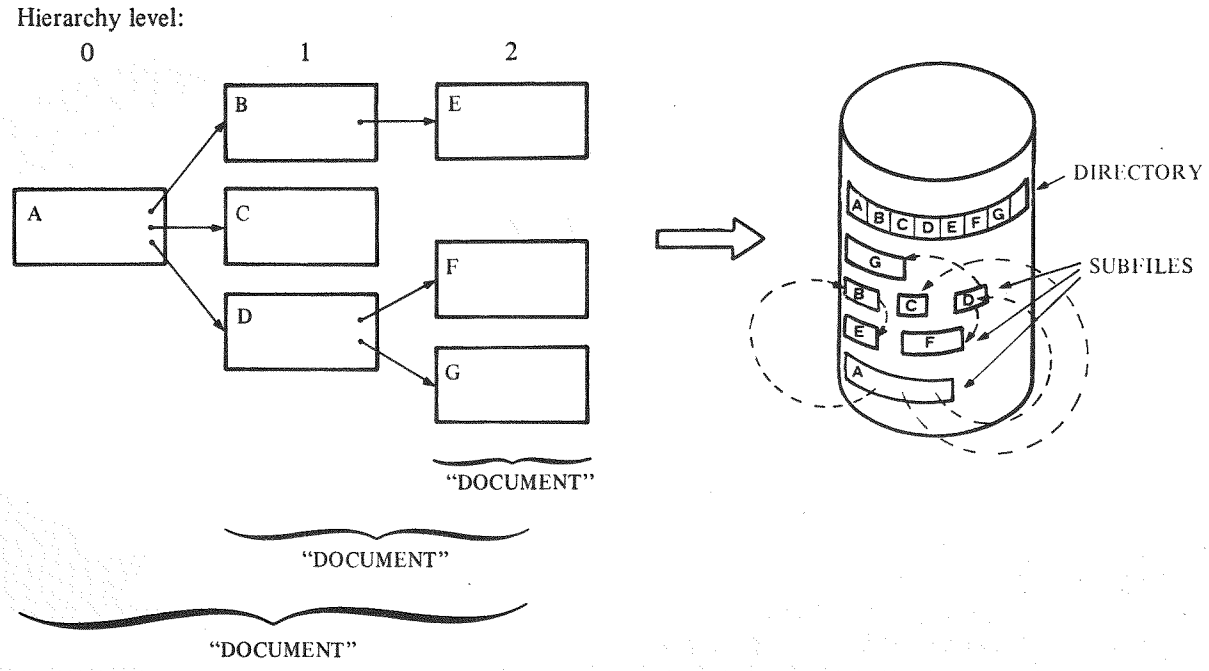


Fig. 5 - The structure of a document

The DOC method must therefore be able to collect these "documentation components" and to assemble them together in the logical way specified by the author.

c) *Documentation structured hierarchically in top/down fashion*

For software products hierarchically structured in top/down fashion (see for example fig. 5) it can be useful for review or training purposes to create issues (versions) of a document which do not contain all the details down to the lowest level before the code level, but which display simply the highest levels of abstractions and definitions.

This capability can be obtained assigning to each documentation block a "hierarchical level" number, which expresses the location of that block in the hierarchy.

An appropriate parameter of DOCUMENTOR will select at run time, in the assembly of the document, only the hierarchical levels specified by the user.

The above points a) b) c) carry to the conclusion that documents are physically implemented on standard source libraries (fig. 5), i.e.:

- a source library is a set of subfiles
- a text is a subfile or a portion of a subfile.

Therefore, a document is constituted of a main subfile (which gives the name and the structure to the document) calling other subfiles of the source library, according to the hierarchical structure.

As shown in fig. 5, a document results in a subfile tree, and each subtree can be considered and printed as a document.

We can consider a "document" a hierarchical structure of text in which:

- each text can refer to other texts through a name (i.e. to other "documents") (see point e below)
- each text can contain (through proper keywords) information about
  - organization of the document (document name, paragraphing) (see point (d) below)
  - editing (commands for spacing, justifying, page ejecting) (see point (f) below)
  - hierarchical level of text parts (indicators for selective printing).

This point of view makes the construction of "documents" in a top/down fashion easy, possibly upgrading them in different moments, tracking through the documentation status the implementation status.

Furthermore, some elementary common texts can be called by different "documents" as "subroutines" (e.g. term definitions, references). Finally, the printout of some specific piece of document can only be obtained with a run-time option.

d) *Capability of creating the table of contents of any document*

This capability, which is common to any document formatting tool, has been implemented in the DOC method by subdividing each documentation block into *paragraphs*, which are considered the *logical units* of documentation.
In the DOC method the table of contents, which is prefixed to the entire document by DOCUMENTOR, is the formatted list of titles and initial page numbers of each paragraph.

e) *Capability of inserting one portion of a text into another*

In the above paragraphs some examples of usage of this capability have been mentioned.

Portions of text which have to be transferred from the original text to another one usually have their own identity and are relatively semantically independent from the original context.

Experience has shown that this case applies to the logical units of text usually considered as paragraphs.

For the above reasons, in the DOC method the decision has been taken to make possible the transfer of entire paragraphs from one text to another.

f) *Document formatting*

A generic, minimal capability of document printout and formatting is required (taking into account that the application is the production of software specifications and not of books or articles for magazines, and that text is in the form of source comments).

3.2. DOCUMENTOR and its commands

A simple documentation language (DOC) has been defined and a tool has been implemented on the Honeywell L62/DPS4 computer to implement the above requirements (see fig. 6). (Ref. (8)).

The processing of source texts performed by DOCUMENTOR is driven by two different kinds of informations:

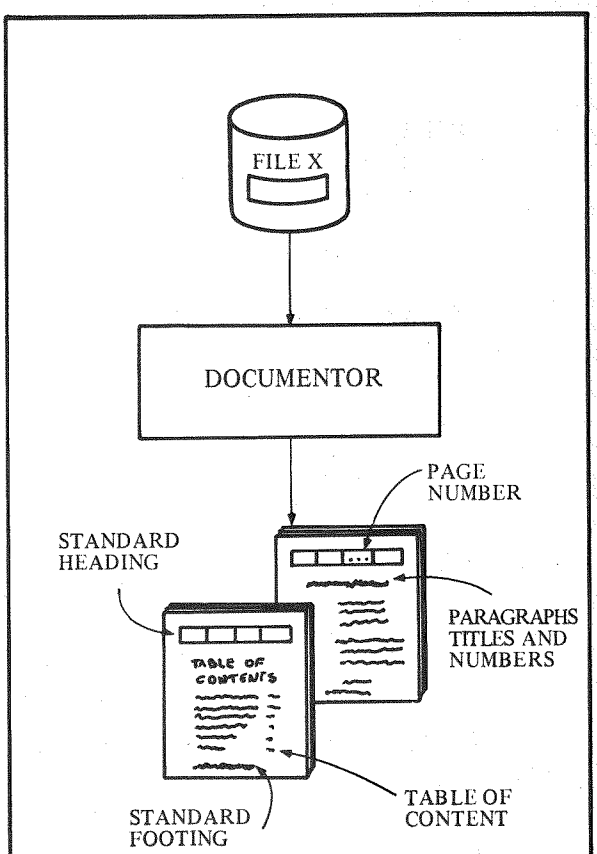


Fig. 6 - Documentor

- the verbs embedded in the source text (devoted to document organization and formatting)

- the parameters of the JCL command to run it (devoted to format change, to maintenance purposes, to selective printing).

3.2.1. DOC embedded verbs

Fig. 7 shows the concise and simple syntax of DOC embedded verbs. These verbs, when properly located within the source text to be printed in a document, will cause the actions described in table 1, which also supplies a reference between verbs and user requirements.

Selection

***$BEGIN DOC** [i]
***$END DOC**

Composition

***$CALL DOC** [paragr. no,] subfile-name

Formatting

***$DOC** [docum. no,] 'title'
***$PAR** [paragr. no,] 'title'
***$TAKE** from-col-no, to-col-no
SPACE [n]
EJECT

Fig. 7 - Embedded verbs

Fig. 8 and 9 give an example of DOCUMENTOR usage. In this example the structure of DOCUMENTOR itself is shown (simplified).
The figures show also how subparagraphing can be obtained.

Fig. 8 shows the texts as they are written by the author.
The text is organized into two levels:

- the former composed simply of a source (SOURCE 1), containing the complete (but empty) structure of the document and some introductory information which is not specific to any component program

- the latter composed of two source texts (SOURCE 2 and SOURCE 3), which define the characteristics of each of the two components in which the program is organized.

The contents of SOURCE 1 are completely enclosed in a BEGIN DOC - END DOC enclosure, defines the paragraph titles and calls the two texts of lower level through the verbs CALL DOC.

The called texts are directly the program listings, with embedded commands: only their documentation parts are enclosed in BEGIN DOC - END DOC pairs. Their paragraphs will become subparagraphs once they are assembled into the main document, inheriting as a prefix the calling paragraph number.

The resulting document is shown in fig. 9.

3.2.2. The DOCUMENTOR Run-time JCL Options

The product is used through the call of a JCL macro.

To be mentioned, for its importance for the end-user in a software production environment, is the parameter *ln*:

- if *ln* is not specified, DOCUMENTOR prints out the text without any references to the source text files

- if *ln* is specified, DOCUMENTOR prints out the text formatted and paged as it was printed for the final user, but the printed text contains in addition:

  — embedded DOCUMENTOR verbs (otherwise skipped)
  — reference to source files name
  — reference to source file lines sequential number

Requirement	DOC verb	Action by DOCUMENTOR
Text blocks inserted into the source programs, but extractable for producing global documents	**SELECTION verbs** Enclosure: *$BEGIN DOC, i «text» *$END DOC	Any text block included in a BEGIN DOC, END DOC enclosure is assembled into the document, by extracting it out of the source code
Documentation of software distributed in more than one source component, and insertion of a portion of a text into another	**COMPOSITION verb** *$CALL DOC parag no, subfile-name	Presence of this verb in a given point of the text causes DOCUMENTOR to make access to the subfile 'subfile-name' and extract from it the portions of text enclosed there between BEGIN DOC, END DOC, inserting them at the CALL DOC place. The called text my contain other calls.
Selective print of top/down documents until a specified hierarchy level = n	*BEGIN DOC i	Documentor selects for printout only the BEGIN DOC, END DOC enclosures where i $\leqslant$ n (n is specified via JCL at run time)
Creation of a table of content	*$PAR parag no, 'title'	This verb, when found in the source text, causes: 1. the paragraph number and titles to be printed out at that position in the document 2. paragraph number, title, page number to be inserted into the table of contents for final printout
Minimal formatting capabilities	*$DOC docum no, 'title' *$TAKE from-col-no, to-col-no SPACE n EJECT	Document number and titles are inserted in the header of each page Only the specified portion of the subsequent lines of text is printed out n blank lines are inserted in the text printed out DOCUMENTOR skips to a new page
Remark: the verbs begin with the characters "*$" in order to reduce scanning costs and to avoid conflicts with the host languages.		

Table 1

This last functionality is of course necessary for the maintenance and updating of the source file: the advantage for the user is that the complete source text is printed in the same manner as the final document, giving the user the possibility of controlling the formatting aspects.

An example of this feature is given in fig. 10.

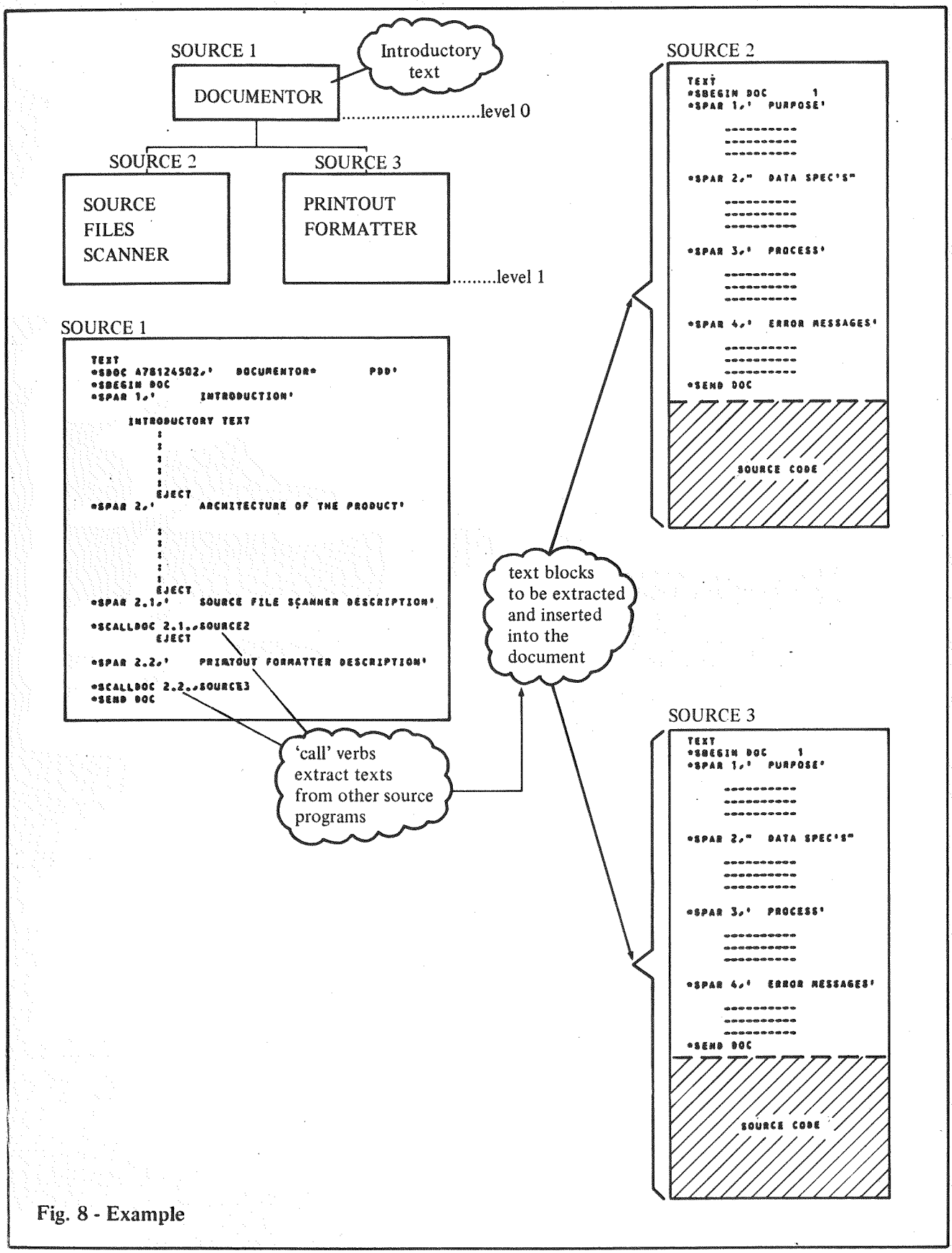


Fig. 8 - Example

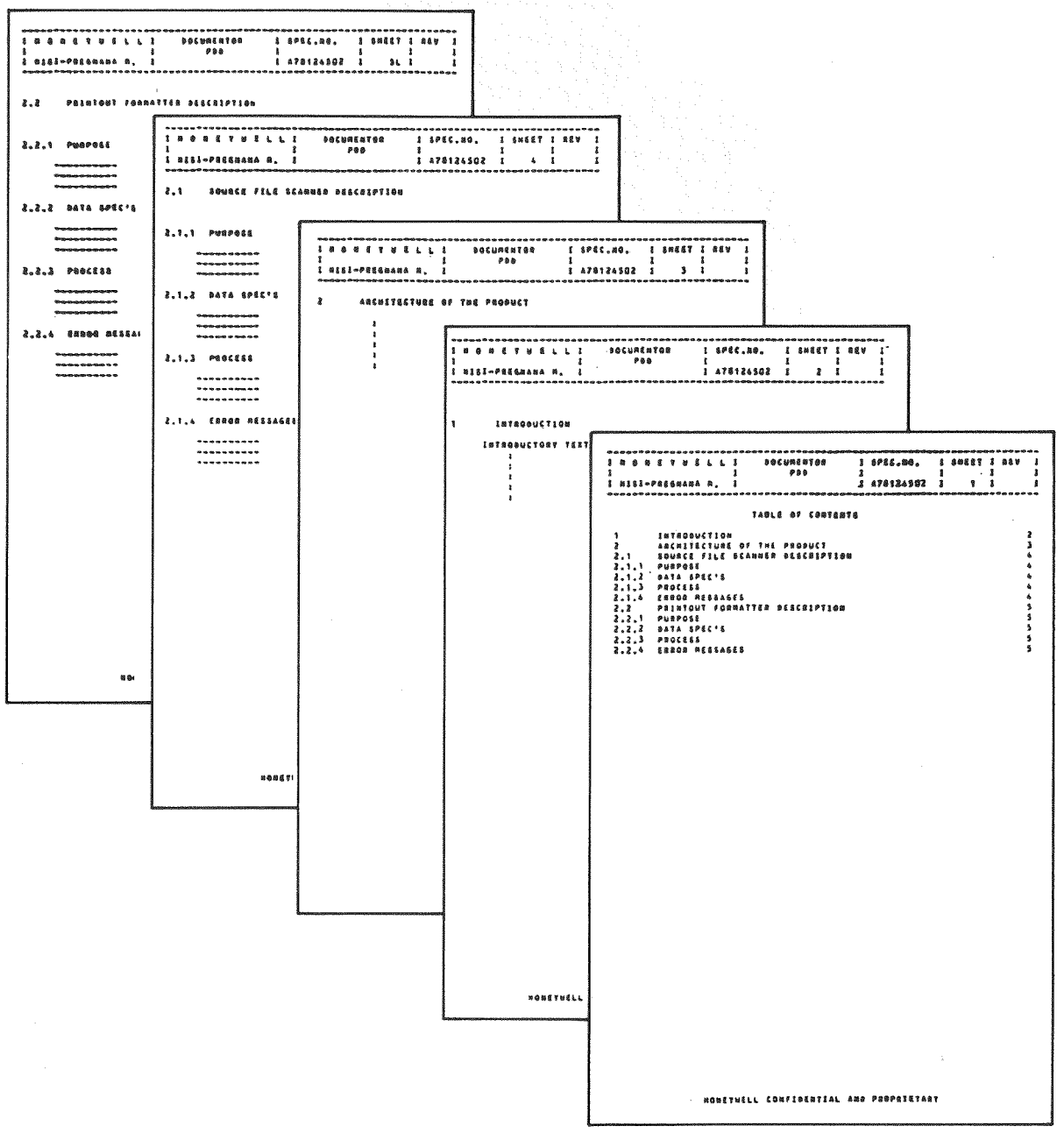


Fig. 9 - The resulting document

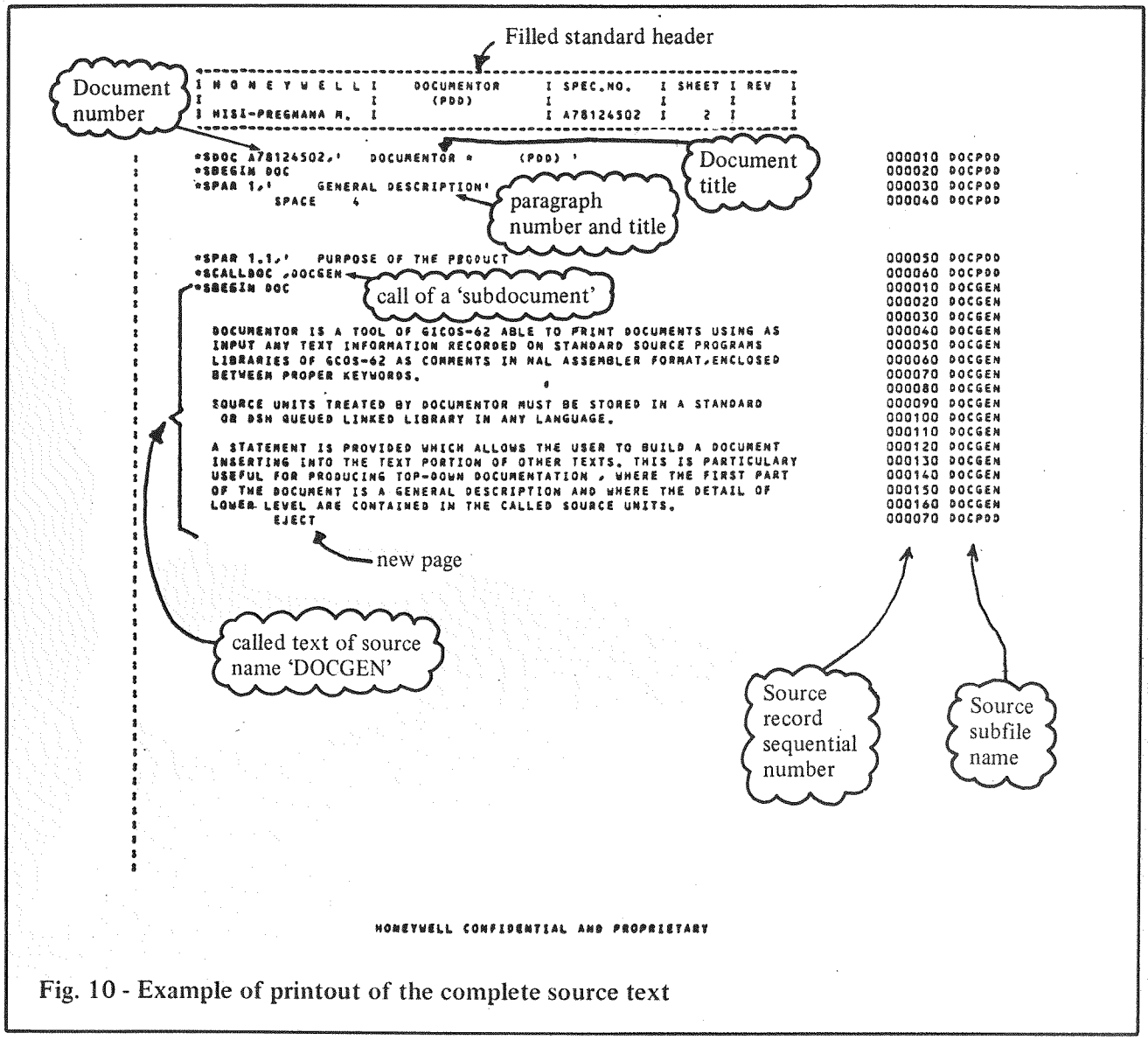


Fig. 10 - Example of printout of the complete source text

Other JCL parameters to be mentioned:

— *level*: if equal to n, DOCUMENTOR prints all text enclosures where BEGIN DOC specifies i $\leqslant$ n. In this way, the proper synthesis of a product description is easily obtainable.

— *Copyrt*: when specified, the user has the possibility of personalizing the header and footing strings printed on each page.

4. "TO DOCUMENT" MEANS "TO IMPLEMENT"

The integration shown above between documentation and program source code (where documentation standards, comment standards and coding standards are strictly coordinated and each one complements the other, and where the method is supported by a production tool) clarifies the fundamental concept that documentation is

— the first step in implementation

— a process continuing with the implementation, as a part of it.

Under this view, each project can be structured at the very beginning, and the source code of every component begins to exist when writing into it its specifications of interfaces with the product and its process description, making in this way much more easier the tracking not only of the functionalities but also of the physical structure.

Once a product has been structured in source components, and this structure has been formalized in the main introductory source text through the proper CALL DOC statements, DOCUMENTOR can be used even if not all source components are yet present in the library: in this case the absence of a component is simply signalled by DOCUMENTOR, without interruption of the production of the document.

5. USAGE OF DOC IN HIS ITALIA SOFTWARE ENGINEERING AND NEXT EXTENSIONS OF THE METHOD AND THE TOOL

Presently DOCUMENTOR is used to produce alla kinds of documents (PFS, EPS, PDD, Test Specs, Quality Reports, etc.).
Taking into account specific aspects of software implementation on Level 62/DPS4, where a large use of macros (common or personalized) is done, and some further common aspects of software documentation, the following extensions are presently being implemented:

— tabulation capability and processing of tables

— personalized documentation of macros. DOCUMENTOR will be able to access the output of the macroexpansion step, in order to insert into the printed documents the effectively expanded code together with the related comments: this capability allows the production of personalized documentation for the specific expansion of a macrocall, and allows the macro definition of parametric documentation.

6. REFERENCES

[1] G. Degli Antoni, G. Torriani, DOCUMENTAZIONE E PRODUTTIVITÀ, Note di Software 3; July/Sept. 1977

[2] G. Gobbi, M. Maiocchi, A GUIDE TO SELF-DOCUMENTED PROGRAMS, Systems Documentation Newsletter, ACM/SIGDOC, Vol. 5 n. 4, Nov. 1978

[3] A. Cicu, SOFTWARE PRODUCT DESIGN DESCRIPTION STANDARD, HISI Doc. n. ZPG 12 (English version)

[4] A. Bicego, SOFTWARE PRODUCT DESIGN DESCRIPTION STANDARD, HISI Doc. n. ZPG 17, June (Italian version)

[5] A. Maserati, S. Mazzocchi, R. Polillo, G. Torriani, DOCUMENTAZIONE PER L'UTENTE: ALCUNE INDICAZIONI E UN ESEMPIO, Note di Software no. 3, July/September 1977

[6] G. Degli Antoni, M. Maiocchi, R. Polillo, PSPN: UN APPROCCIO AI PROBLEMI DI COMUNICAZIONE DELL'INFORMAZIONE TECNICA, Note di Software, 6/7

[7] A. Chiapparino, DOCUMENTOR USER GUIDE, HISItalia Software Engineering Internal Manual, Doc. order n. A78124506, June 1979

NOTE SULL'USO DI FUNZIONALITÀ COBOL IN AMBITO PHOS

C. Magnoni(°) - E. Spoletini(°*) - E. Toscani(°)

Riassunto

Scopo dell'articolo è illustrare come l'uso di funzionalità standard disponibili in un linguaggio di programmazione (es. Cobol) sia possibile anche in questo ambito metodologico. Infatti un corretto utilizzo di tali strumenti implementativi è possibile purchè il loro utilizzo non sia condizionante in fase di disegno ma solo in fase di realizzazione.

Abstract

The aim of this paper is to show how it is possible, also in methodological environment, to use standard functionalities available in a programming language (e.g. Cobol). In fact, a correct utilization of these implementary tools is possible if their use doesn't affect the design phase but only the implementation phase.

1. INTRODUZIONE

Chi adotta un approccio strutturato può avere l'impressione di dover rinunciare ad alcune prestazioni offerte dal software, in quanto possono apparire contrastanti le linee guida della metodologia utilizzata e gli automatismi impliciti negli strumenti software disponibili.

Non è detto che questo giudizio sia completamente falso, infatti esistono delle circostanze che lo convalidano: si tratta, comunque, o di atteggiamenti errati o di carenze metodologiche. Può, infatti, verificarsi il caso in cui le linee guida dettate dalla metodologia siano deboli, quindi l'atteggiamento di chi programma diventa incerto e risulta allora facile confondere le funzionalità di una porzione di software con vincoli per l'applicazione che si sta costruendo. Può, inoltre, intervenire un tipico errore di valutazione sull'uso di certi strumenti software che devono essere presi in considerazione al momento dell'implementazione ed essere utilizzati solo quanto sono veramente utili, ... non a tutti i costi. Un altro pericolo che si può verificare è dovuto al fatto che certe funzionalità del software consentono di ottimizzare la stesura di una porzione di codice; ciò può far dimenticare che il primo approccio al problema deve essere di tipo logico, mentre la fase di ottimizzazione è solo successiva: questo per evitare che diventi più propriamente una fase di "distruzione" della struttura del programma. In questo articolo, ovviamente non a dimostrazione, ma a sostegno di questa tesi, presentiamo un programma che sarà implementato in Cobol avanzato (sfruttando il Sort interno e la Report writer), dopo essere stato disegnato seguendo i passi previsti dalla metodologia PHOS: sarà evidente che non si genereranno contrasti nè situazioni anomale quando saranno utilizzati strumenti sofisticati in un rigoroso contesto metodologico.

Nello svolgimento dell'esempio conduttore, allo scopo di facilitarne la lettura, richiameremo in modo sintetico le linee guida, i supporti concettuali e linguistici ed i passi previsti da PHOS.

2. CARATTERISTICHE DI PHOS

La metodologia PHOS consente di ottenere programmi di buona qualità, con caratteristiche standard di documentazione e codifica, mediante un procedimento definito con precisione.
Inoltre, queste caratteristiche sono conservate completamente anche dopo una serie di interventi di manutenzione.

Alla definizione della struttura del programma si arriva, secondo la metodologia PHOS, attraverso la definizione delle strutture dei dati, che vengono costruite individuando le unità logiche, cioè l'insieme delle informazioni che caratterizzano un certo soggetto di trattamento nell'ambito del problema. In questo contesto, diventa diverso il ruolo assunto dai dati e diversa la filosofia di costruzione del programma.

Infatti, i dati non sono semplicemente quegli oggetti su cui il programma eseguirà un numero finito di trasformazioni, ma assumono il ruolo di guida nella stesura del programma. In tale contesto, il programma è semplicemente un insieme di moduli che realizzano il trattamento di ogni unità logica di output (nel caso in cui le strutture di INPUT e OUTPUT siano tra di loro coerenti).

La costruzione del programma non si riduce più ad una sorta di invenzione di una funzione di trasformazione dell'input nell'output, ma diventa il risultato di un insieme di fasi rigidamente sequenziate che, a partire dalle specifiche del problema, porta-

(°) Honeywell I.S.I. - Formazione
(*) Università di Milano - Ist. di Matematica

no, attraverso le strutture dei dati, alla definizione del programma in termini di "composizione di trattamenti" organizzati per livelli gerarchici successivi.

Il criterio con il quale si affronta il processo di costruzione del programma è rigorosamente top-down in ogni sua fase; si individuano, cioè, all'interno delle strutture di I/O le unità logiche fondamentali del problema e, livello per livello, attraverso un processo di raffinamento, si procede alla scomposizione, fino ad evidenziare le unità logiche elementari accessibili con operazioni di I/O, cioè i records logici; le caratteristiche delle strutture dei dati si riflettono su quella del programma che da esse direttamente deriva.

Nella programmazione tradizionale, proprio perchè sono realtà diverse, dati e programma sono descritti con linguaggi diversi: questo è uno dei motivi che hanno portato a considerarli come entità disgiunte non direttamente collegabili se non dal punto di vista funzionale.

Nell'ambito della metodologia PHOS, è stato definito un linguaggio in grado di descrivere in modo omogeneo oggetti appartenenti a realtà diverse, un linguaggio che permette di descrivere i dati e i programmi in termini di struttura logica e organizzativa rispettivamente delle informazioni e dei trattamenti.

Gli strumenti concettuali messi a disposizione costituiscono una guida ad un esame dell'analisi, ad uno studio critico del modo in cui il problema deve essere descritto e affrontato, e alla individuazione di entità concettuali presenti nel problema che possono risultare inespresse nell'analisi.

L'attenzione del programmatore è portata su aspetti di esame del problema dal punto di vista dell'architettura generale del programma, ridimensionando gli aspetti di coerenza, di codifica e di sequenza di esecuzione che, frazionati in moduli elementari, all'interno di una ben precisa struttura, perdono automaticamente la loro complessità.

3. SUPPORTI CONCETTUALI LINGUISTICI

Precisiamo ora il significato della terminologia più corrente nella metodologia PHOS.

- **struttura**
  è uno schema di rappresentazione secondo regole di definizione e di associazione che rispettivamente permettono di individuare gli elementi della struttura e ne stabiliscono le relazioni reciproche;

- **flusso logico**
  è l'organizzazione che è possibile individuare all'interno di un insieme di dati in base alle specifiche del problema, in base, cioè, a come i dati di quell'insieme intervengono nel processo di elaborazione.
  Sono individuati due tipi di flussi logici:

  — flusso logico sequenziale (FS)
  è un insieme di unità logiche che intervengono nel processo di elaborazione secondo una sequenza ben precisa definita a priori;

  — flusso logico random (FLR)
  è un insieme di unità logiche ognuna delle quali interviene nell'elaborazione in modo indipendente dalle altre;

- **unità logica**
  è l'insieme delle informazioni relative ad uno stesso soggetto, cioè l'insieme delle informazioni che subisce un ben preciso trattamento;

- **unità di trattamento**
  è l'insieme delle istruzioni che esauriscono il trattamento di una unità logica.

Per definire e scomporre completamente una unità logica o di trattamento useremo le tre strutture di controllo elementari (fig. 1):

- **sequenza**
  una unità logica è definita come una sequenza se è costituita da un insieme ordinato di unità logiche di tipo diverso;

- **selezione**
  una unità logica è definita come una selezione fra due o più unità logiche se può essere costituita da una oppure dall'altra sulla base del verificarsi o meno di una determinata condizione;

- **iterazione**
  una unità logica è definita come iterazione di unità logiche se è costituita da una sequenza di un numero imprecisato di unità logiche dello stesso tipo.

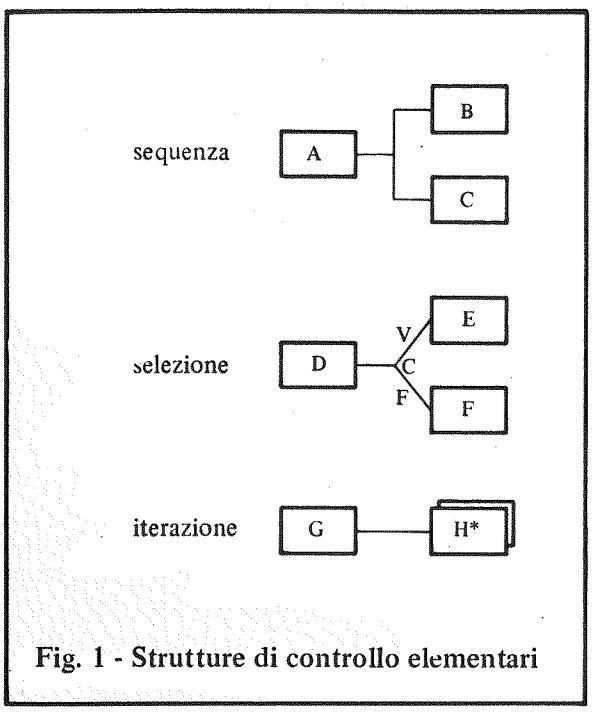


Fig. 1 - Strutture di controllo elementari

4. L'ESEMPIO CONDUTTORE

- **Organigramma**

  — vedi fig. 2.

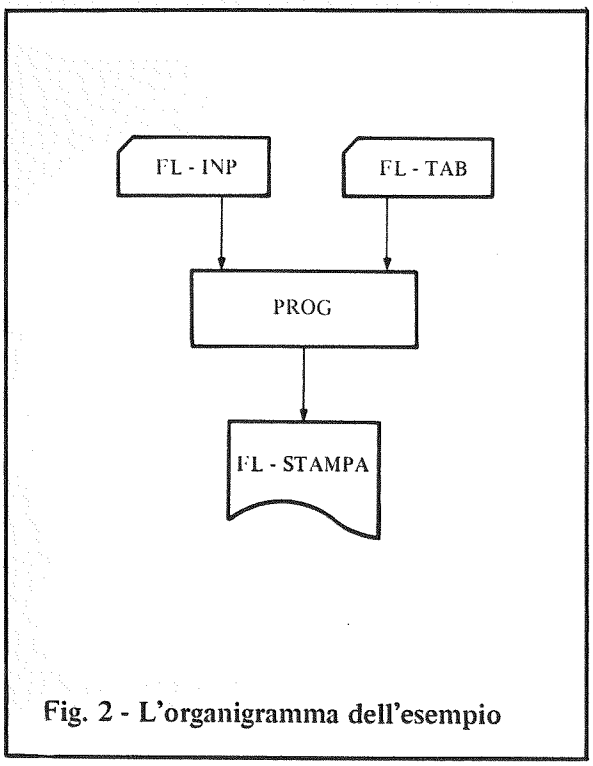


Fig. 2 - L'organigramma dell'esempio

- **i files**

FL-INP — flusso ad organizzazione sequenziale: flusso movimenti ordinato per filiale e cliente

FL-TAB — flusso ad organizzazione sequenziale: tabella filiali ordinata per codice filiale

FL-STAMPA — tabulato conti correnti per filiali con riepilogo filiali come da tracciato.

- **specifiche di elaborazione**

Caricare la tabella in memoria.
All'inizio del tabulato stampare una pagina di titoli.
Scorrere sequenzialmente il flusso FL-INP
  scartare i record movimento con TSK ≠ m
  per ogni record movimentato con TSK = = m stampare i campi:
    DATA MOV, DESCR-MOV e IMP-MOV
  per ogni nuovo cliente andare a pagina nuova
    stampare un'intestazione cliente
    a fine cliente stampare il totale degli importi del cliente
  per ogni nuova filiale andare a pagina nuova
    stampare un'intestazione filiale
    cercare la filiale in tabella
    a fine filiale stampare il totale degli importi della filiale
      se la filiale è presente in tabella
        aggiornare il campo IMP-TOT
        sommandovi il contenuto
        di IMP-MESE-CORR
        inserire nel campo
        IMP-MESE-CORR il totale
        degli importi della filiale
      se la filiale è assente
        aggiornare il campo
        IMP-ALTRE-FIL (in W/S)
        sommandovi il totale degli
        importi della filiale
  a fine flusso stampare a pagina nuova il totale generale degli importi di tutte le filiali
    stampare a pagina nuova la tabella filiali (con intestazione) ordinata per IMP-TOT crescente. Alla fine della tabella stampare il totale IMP-ALTRE-FIL come da tracciato.

5. SVILUPPO DELL'ESEMPIO CONDUTTORE

Procediamo ora allo sviluppo del programma utilizzando come esempio, passo passo nel corso della presentazione, i passi logici previsti da PHOS.

a) Analisi e descrizione gerarchica dei dati d'ingresso e di uscita

I dati vengono descritti mediante raffinamenti top-down che ne dettagliano completamente la struttura: ogni elemento di un dato livello può essere specificato come sequenza, alternativa o iterazione di elementi di livello inferiore.

Mentre i diversi files d'ingresso vengono descritti separatamente, tutti i risultati, anche se fisicamente distribuiti su differenti supporti (o addirittura non presenti fisicamente), vengono descritti in un'unica struttura, che sarà il punto di partenza per la deduzione della struttura del programma. In tal modo, la struttura del programma sarà in grado di tener conto in modo esaustivo, e secondo un ordine stabilito, di tutti gli obiettivi che si vogliono raggiungere.

Allo scopo di individuare tutti gli elementi logici che intervengono nel problema si utilizza una sola regola di composizione per volta.

Il risultato di questa operazione mette in evidenza la struttura logica dei dati da trattare e quindi le entità concettuali che rappresentano livelli riassuntivi di insiemi di dati più semplici (che non sono quasi mai espresse nell'analisi). In questo modo, la descrizione dei dati tiene conto non solo della successione fisica degli elementi da emettere (o non) ma anche dell'ordine temporale con cui i vari flussi vanno riempiti.

Chiamiamo OUT la struttura complessiva dei risultati da conseguire: OUT è un flusso logico sequenziale (FLS).

Al contrario, la descrizione dei flussi di input è separata in quanto sarà proprio l'esecuzione del programma a determinare la corretta sequenzializzazione delle informazioni presenti su più flussi.
In INP, complesso delle descrizioni dei flussi in ingresso, saranno evidenziati gli elementi strutturali rilevanti per i singoli flussi e le correlazioni reciproche.

Nella costruzione del programma è importante, al fine di un semplice inserimento delle funzioni di I/O, conoscere quali degli elementi logici individuati corrispondono ad una entità fisica reale, cioè ad un record: ciò viene indicato (con una R) sulla struttura dei dati.

In fig. 3 sono evidenziate le strutture di OUT e INP relative al problema proposto. Notare che il flusso tabella è utilizzato in questo caso sia come FLR (per l'aggiornamento del campo IMP-TOT) che come FLS (per la stampa del suo contenuto in ordine di importo crescente).

b) Deduzione della struttura del programma

La struttura del programma si ottiene, in prima approssimazione, semplicemente ricopiando la struttura dei risultati, interpretandola via via come l'insieme dei "trattamenti" che consentono di generarli, ed anteponendo e posponendo, a quelle unità che vengono raffinate al livello successivo, un blocco di inizio ed uno di fine elaborazione (con lo scopo di risolvere problemi di interfaccia con il livello gerarchico precedente).

La struttura dei dati di uscita costituisce, pertanto, il modello per la deduzione della struttura del programma.

Lo strumento grafico utilizzato in questa fase è lo stesso impiegato per la descrizione delle strutture dei dati ed è concettualmente identico.
In questa fase devono essere direttamente specificate le condizioni di selezione e quelle d'uscita dalle iterazioni. Vedere in fig. 4 la struttura di programma dedotta per l'esempio conduttore.

c) Analisi della consistenza tra strutture di ingresso e di uscita

Il programma, costruito a partire dalla struttura dei risultati, può essere mantenuto con tale struttura solo se la forma con cui si presentano i dati in ingresso è coerente con quella dei risultati prodotti.

Questa fase ha lo scopo di evidenziare eventuali difficoltà ("conflitti") in tal senso.
L'individuazione dei conflitti tra le strutture dei dati viene fatta analizzando le strutture dei dati di ingresso e di uscita, livello per livello.

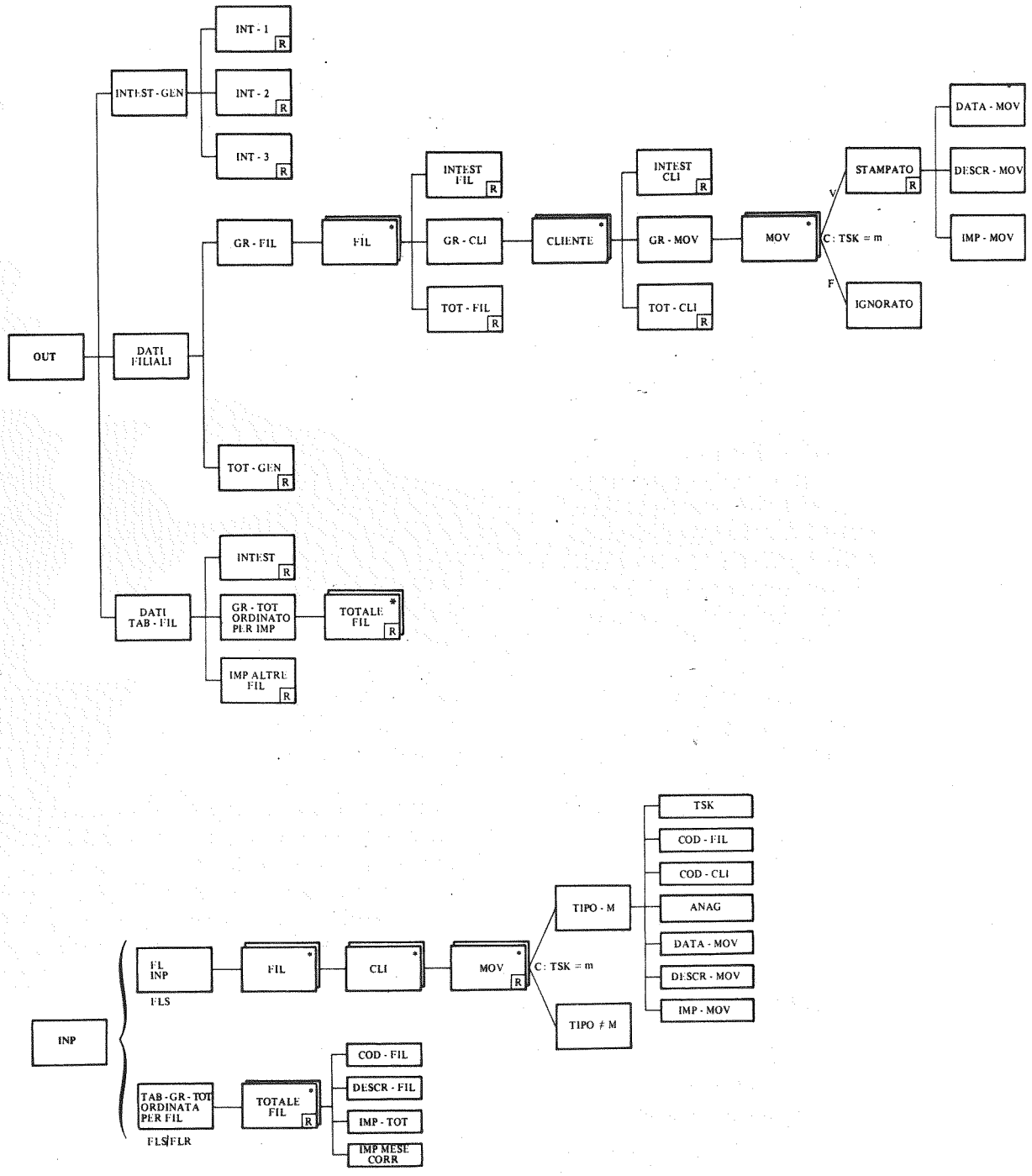


Fig. 3 - Le strutture dei dati dell'esempio conduttore

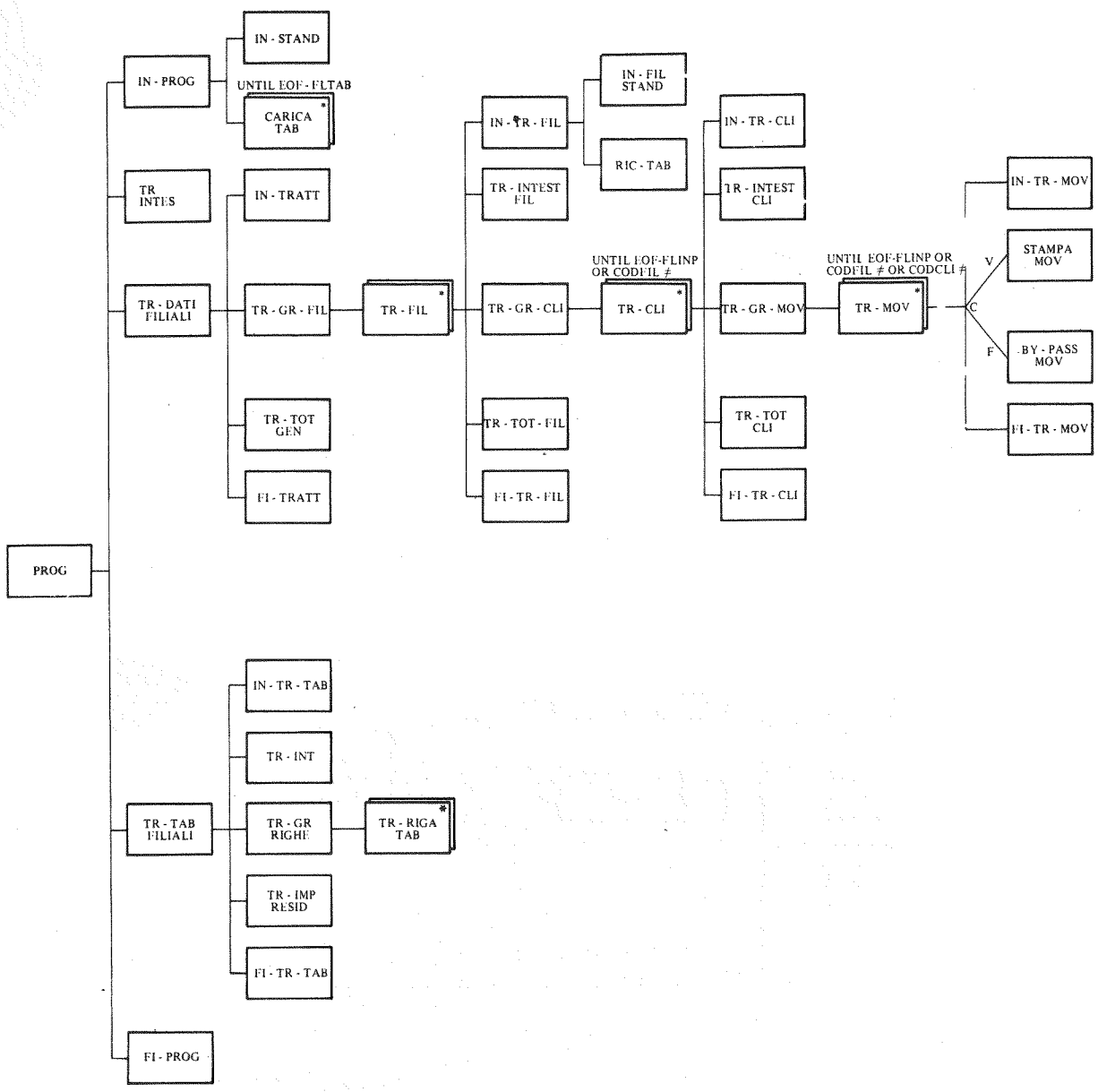


Fig. 4 - PROG dedotto, in prima approssimazione, da OUT di fig. 3

I conflitti sono classificati (fig. 5) in:

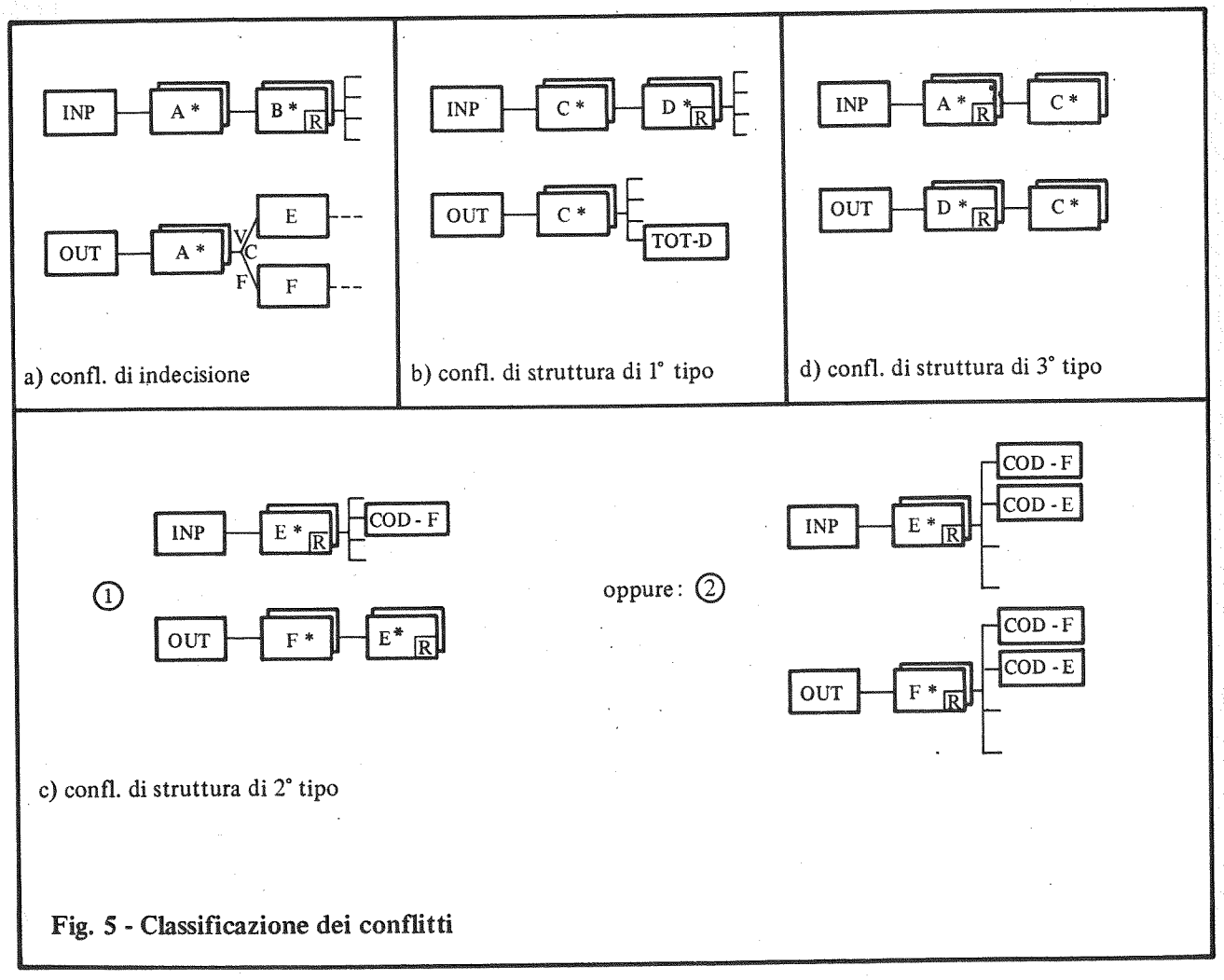
- **conflitti di indecisione**, nel caso in cui la mancata corrispondenza tra le strutture avvenga a livello di una struttura alternativa presente in output e non in input

- **conflitti di struttura**, nel caso in cui la mancanza di corrispondenza sia a livello di strutture iterative.
  Vengono individuati tre tipi di conflitti di struttura:
  - — 1° tipo : un'iterazione presente in input è assente in output
  - — 2° tipo : un'iterazione presente in output manca in input oppure la stessa unità logica iterativa è presente sia in input che in output ma ordinata in chiave diversa
  - — 3° tipo : iterazioni presenti in input mancano in output e viceversa.

È importante notare che i conflitti di terzo tipo non sono banalmente la composizione di conflitti di primo e secondo tipo ma trattasi di situazioni provenienti ad esempio da differente organizzazione fisica dei dati sui supporti (ad esempio elaborazione "di parole" con input da telescrivente che ha il "carattere" come unità fisica d'ingresso).

Tale classificazione di conflitti permette di fornire precise indicazioni su come intervenire in relazione al tipo di conflitto riscontrato.

Per quanto riguarda l'esempio proposto, in cui il confronto viene realizzato tra strutture di OUT e strutture di FL-INP e TABELLA (inteso come FLS), si riscontra la presenza di un conflitto di struttura del 2° tipo, causato dal diverso ordinamento dei dati di tabella in INP (ordinamento per codice filiale) e in OUT (ordinamento per importi crescenti).



a) confl. di indecisione

b) confl. di struttura di 1° tipo

d) confl. di struttura di 3° tipo

① ②

c) confl. di struttura di 2° tipo

Fig. 5 - Classificazione dei conflitti

Si noti come, nel corso del confronto, non siano state tenute in considerazione alcune unità logiche presenti in OUT e non in INP (INTES-FIL, TOT-FIL, TOT-GEN, ecc.): queste sono infatti definite banali e la loro presenza non genera alcun conflitto. I conflitti significativi tra le strutture, come abbiamo visto, si presentano infatti in corrispondenza di strutture iterative e/o alternative.

d) Modifiche della struttura del programma

La presenza di conflitti, evidenziata dall'analisi descritta al punto precedente, indica che la struttura costruita è adeguata per rappresentare il progetto di un "produttore di risultati" ma inadeguata a renderlo anche "lettore di dati".

Si pone, allora, il problema di adeguare la struttura originaria apportando opportune modifiche: l'intervento è realizzato senza difficoltà in quanto le tecniche di risoluzione sono classificate parallelamente alla classificazione dei tipi di conflitto riscontrati.

È importante rilevare che le modifiche imposte in queste situazioni non sono rifacimenti di parti già pronte, ma implementazioni della struttura del programma dedotte dalla struttura dell'input:

- *Nel caso di conflitti di indecisione*, viene dettagliato il blocco d'inizio trattamento, relativo al livello gerarchico in cui si presenta il conflitto, in una sequenza di due blocchi: il primo raccoglie le operazioni elementari standard, il secondo le operazioni richieste per il superamento del conflitto, cioè le operazioni che permettono la preparazione della condizione (fig. 6).

- *Nel caso di conflitti di struttura*, in corrispondenza dei tre sottocasi visti si identificano i seguenti tre tipi di intervento:

  — *nel caso di iterazioni assenti in output*, si inserisce nel blocco iniziale del livello in cui si presenta il conflitto una elaborazione di quelle unità logiche di input iterate (fig. 7);

  — *nel caso di conflitti del 2° tipo*, cosa che presuppone un differente ordinamento (anche solo parziale) tra ingresso e uscita, si inserisce nel blocco iniziale del livello individuato un blocco che permette di ottenere l'ordinamento richiesto (fig. 8);

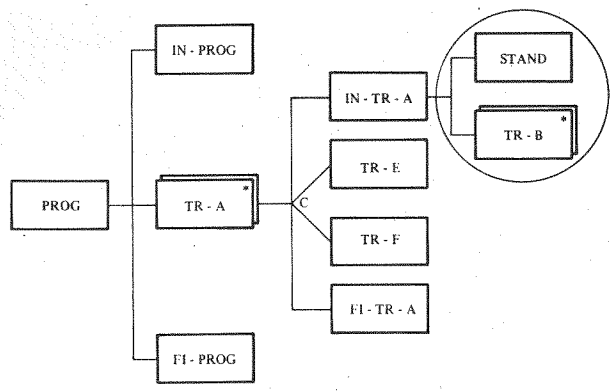


Fig. 6 - Tecnica di risoluzione per conflitti di indecisione

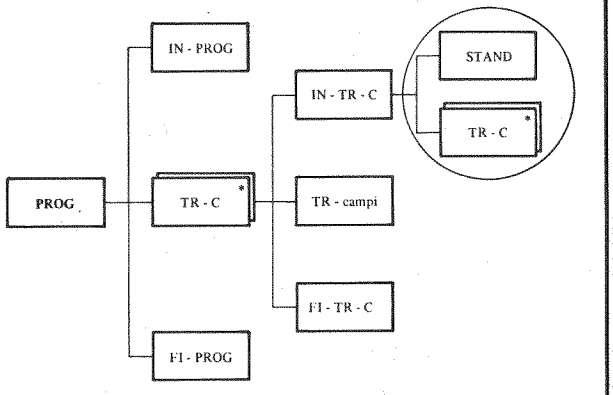


Fig. 7 - Tecnica di risoluzione per conflitti di struttura di 1 tipo

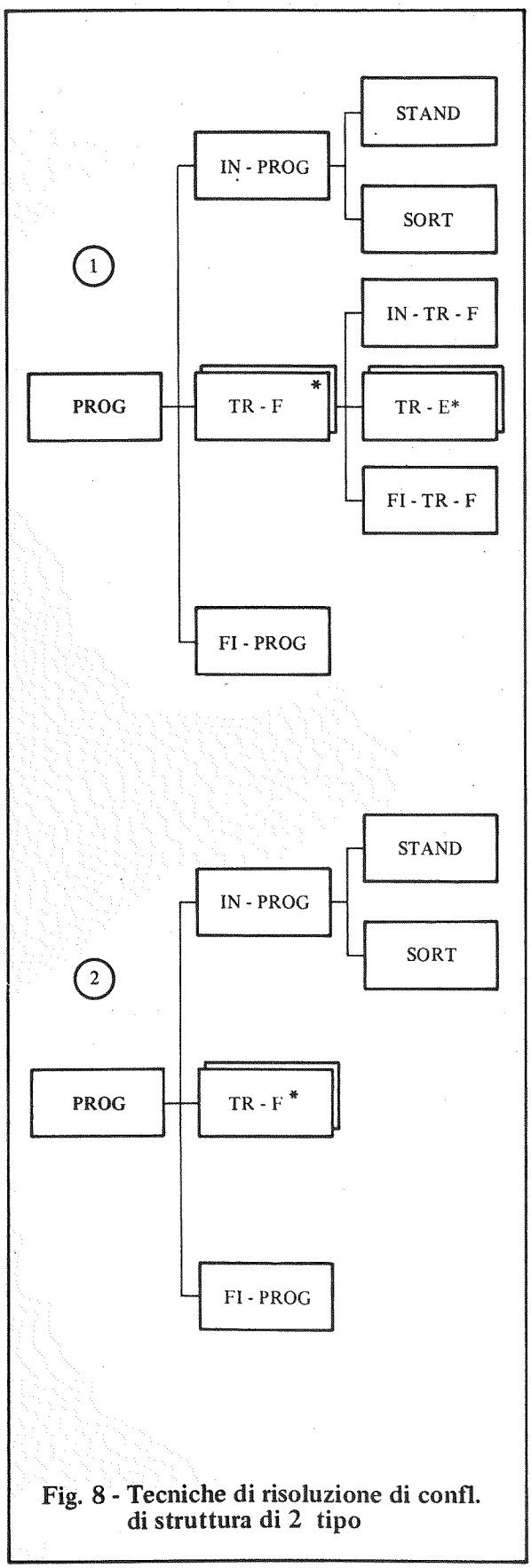


Fig. 8 - Tecniche di risoluzione di confl. di struttura di 2 tipo

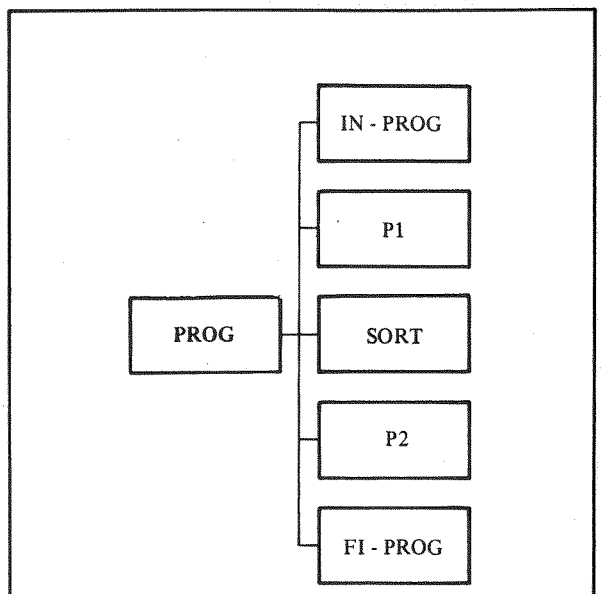


Fig. 9 - Tecniche di risoluzione di confl. di struttura di 3 tipo

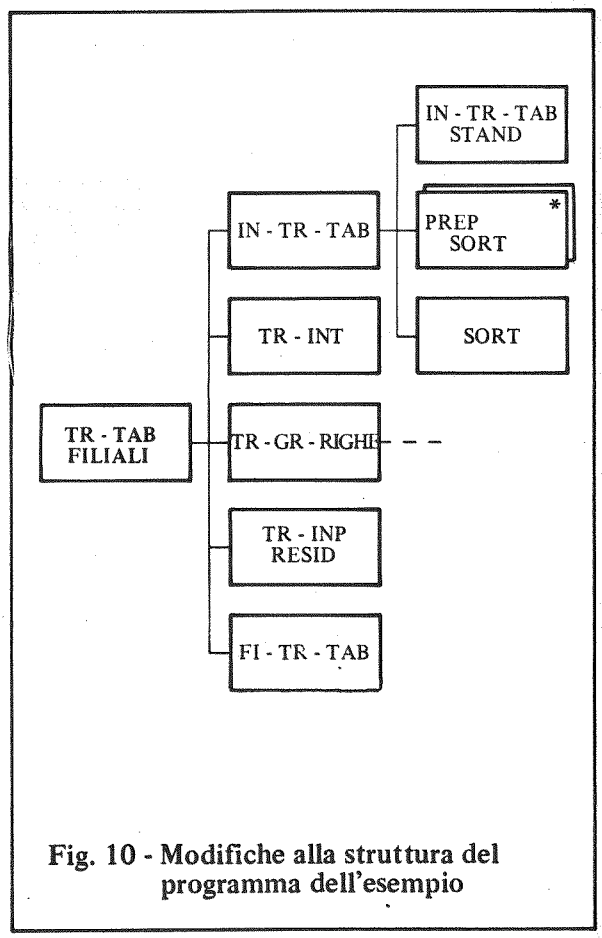


Fig. 10 - Modifiche alla struttura del programma dell'esempio

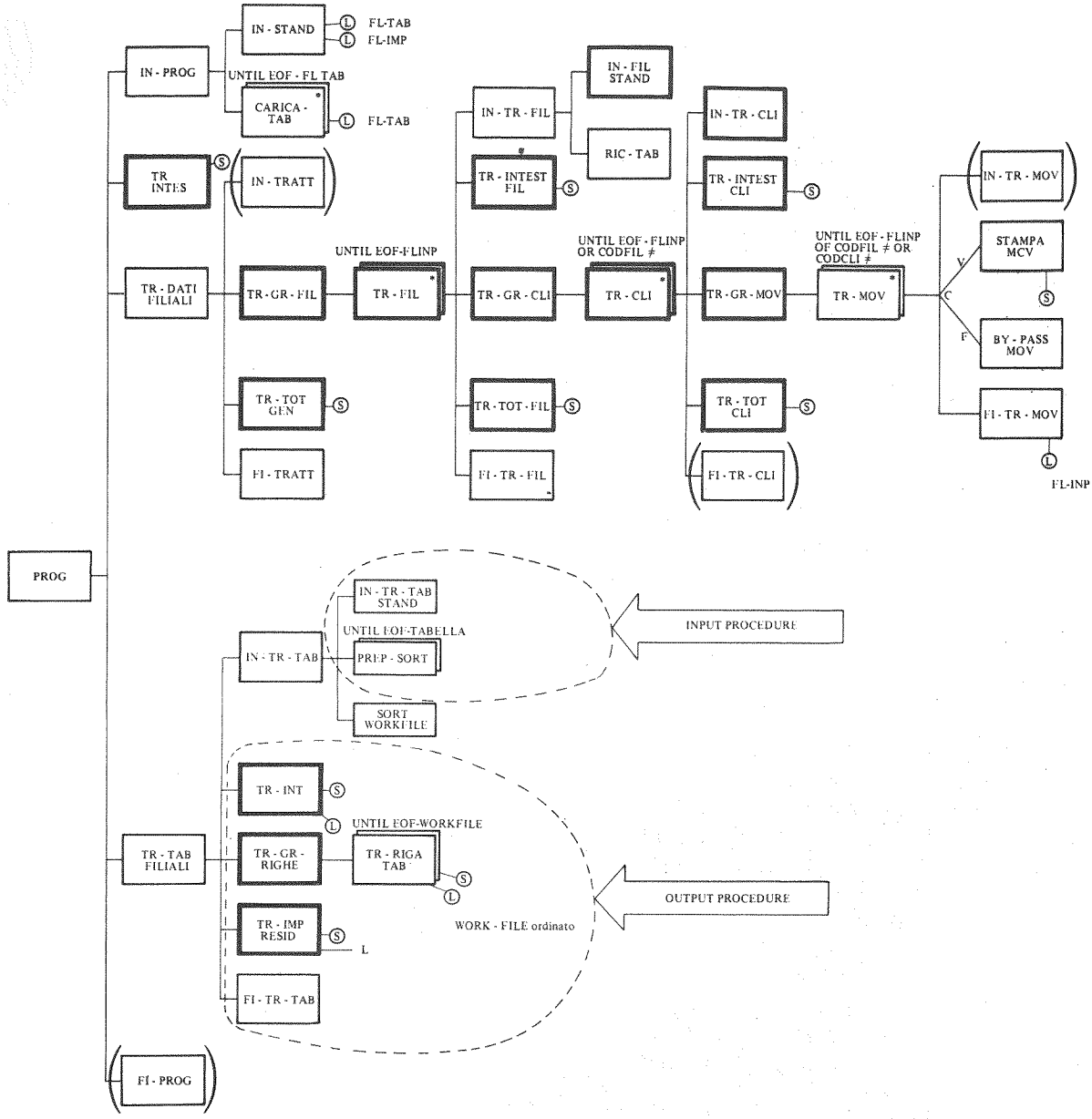


Fig. 11 - Struttura completa del programma con evidenziati i moduli gestiti dalle Report ▭ e quelli vuoti (▭)

- *nel caso di iterazioni assenti sia in input che in output*, si indica la scomposizione del livello del programma (in cui si individua il conflitto) in tre parti distinte: la prima ha il compito di spezzare le unità fisiche di input in sotto-unità necessarie alla costruzione dell'output, la seconda ha il compito di ordinare i dati così costruiti secondo un ordine opportuno; la terza, partendo dai dati opportunamente trasformati, ha il compito di produrre i risultati richiesti (fig. 9).

La fig. 10 mostra la modifica apportata alla struttura del programma relativa all'esempio proposto. Tale modifica consiste nella scomposizione del modulo IN-TR-TAB in un modulo standard (in cui saranno realizzate le normali operazioni previste in assenza di conflitto) e in un modulo che realizza l'ordinamento. Nel caso specifico questo modulo, SORT-WORK-FILE, deve essere preceduto dalla preparazione del file da ordinare a partire dai dati in tabella.

e) Inserimento delle operazioni elementari

Una volta definita la struttura del programma, ci si trova di fronte ad unità di trattamento organizzate gerarchicamente. A queste unità saranno associate le istruzioni elementari, che verranno poi tradotte nel linguaggio di programmazione desiderato, secondo un procedimento rigorosamente top-down sulla base di precise indicazioni fornite dalla metodologia.

Prima di procedere all'inserimento delle operazioni elementari si individuano, utilizzando apposite regole, i moduli in cui devono essere inserite le operazioni di I/O e si procede alla loro visualizzazione sulla struttura del programma.

In fig. 11 è mostrato, relativamente all'esempio proposto, l'inserimento, sulla struttura del programma, delle operazioni di I/O.

f) Codifica

La metodologia non è legata ad uno specifico linguaggio di programmazione, tuttavia è innegabile che meglio si utilizza in tutti i suoi aspetti con linguaggi di programmazione che offrano le strutture di controllo della programmazione strutturata e facilitino l'organizzazione gerarchica dei programmi. La codifica dovrà, sostanzialmente, essere divisa in due parti: la prima consiste nella codifica, livello per livello, dell'albero del programma che, al pari di un "monitor", richiama tutte le routines corrispondenti alle foglie dell'albero; la seconda parte consiste poi nella codifica di queste ultime. Una codifica di questo tipo è particolarmente utile in quanto il programma risulta già predisposto per la segmentazione.

È ovviamente questa la fase in cui possono intervenire elementi caratteristici del linguaggio di programmazione ad alleggerire o appesantire l'intera fase. Compito del programmatore sarà fare un'esatta valutazione sull'uso, o meno, dei tools disponibili.
Per quanto riguarda la codifica dell'esempio proposto, osserviamo che sono utilizzabili due tools che possono facilitarne la scrittura: il sort interno e la Report Writer.

Per il sort, il vantaggio consiste nell'evitare di spezzare il programma in due parti distinte: con una opportuna scelta di Input e Output Procedure (vedi fig. 11 e tabulato) si può integrare la 1ª parte del programma con quella di stampa usando il Sort a cui la prima parte affida i records da ordinare con una Release e da cui la seconda parte riceve i records ordinati tramite una Return.

Inoltre, osserviamo che l'uso delle Report permette di non descrivere in Procedure Division alcune porzioni dell'albero: precisamente quelle caratterizzate (in fig. 11) da un bordo più marcato.

A livello di codifica abbiamo evidenziato l'intera struttura dell'albero (a scopo di commento) come sarebbe risultata secondo tecniche standard di codifica in assenza di tali strumenti (vedere il tabulato relativo): questo consente di mantenere una chiara visibilità della logica del prodotto.

6. RIFERIMENTI

[1] T. Bettinazzi, M. Maiocchi, A. Maserati, F. Monzani, E. Spoletini, E. Toscani, PHOS: UNA METODOLOGIA PER LA COSTRUZIONE VELOCE ED AFFIDABILE DI PROGRAMMI, NOTE DI SOFTWARE n. 4 (ottobre-dicembre 1977)

[2] T. Bettinazzi, A. Maserati, F. Monzani, E. Toscani, PHOS: UNA APPLICAZIONE IN AMBIENTE EDP, NOTE DI SOFTWARE n. 4 (ottobre-dicembre 1977)

[3] T. Bettinazzi, A. Maserati, F. Monzani, E. Spoletini, E. Toscani, VANTAGGI E CARATTERISTICHE DELLA METODOLOGIA PHOS, PARTE I: CONCETTI BASE E CONFRONTO CON LE TECNICHE DI PROGRAMMAZIONE TRADIZIONALI, Management e Informatica, Dicembre 1979; PARTE II: METODO JACKSON E METODOLOGIA PHOS, Management e Informatica, febbraio 1980; PARTE III: CONFRONTO CON LA METODOLOGIA WARNIER E CONSIDERAZIONI CONCLUSIVE, Management e Informatica, aprile 1980

La metodologia PHOS è stata sviluppata nell'ambito del CP Project, progetto di cooperazione fra Honeywell Information Systems Italia e Università di Milano, Istituto di Cibernetica.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPWR-XX.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. LEVEL-64.
OBJECT-COMPUTER. LEVEL-64.
SPECIAL-NAMES. DECIMAL-POINT IS COMMA
               OBJECT IS COMMA.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FL-INP ASSIGN TO SKEDE.
    SELECT FL-TAB ASSIGN TO SKTAB.
    SELECT FL-STAMPA ASSIGN TO STA-SYSOUT.
    SELECT FL-SORT ASSIGN TO H-SORT.
DATA DIVISION.
FILE SECTION.
FD FL-INP
    LABEL RECORD OMITTED
    DATA RECORD REC-IN CONFLICT.
01  CONFLICT  PIC X(264).
01  REC-IN.
    02 TSK PIC X.
    02 COD-FIL PIC X(4).
    02 COD-CL PIC X(5).
    02 ANAG     PIC X(40).
    02 MOVIM.
       03 DATA-M PIC X(8).
       03 DESCR PIC X(10).
       03 IMP   PIC 9(7).
FD FL-TAB
    LABEL RECORD OMITTED
    DATA RECORD REC-TAB.
01  REC-TAB.
    02  COD-FIL-F PIC X(4).
    02  DESCR-FIL-F PIC X(15).
    02  IMP-TOT-F PIC 9(12).
    02  IMP-MESE-CORR-F PIC 9(9).
FD FL-STAMPA
    LABEL RECORD OMITTED
    REPORTS ARE TABUL-CC TABUL-FIL.
SD FL-SORT
    DATA RECORD REC-SORT.
01  REC-SORT.
    02 COD-FIL-S PIC X(4).
    02 DESCR-FIL-S PIC X(15).
    02 IMP-TOT-S PIC 9(12).
    02 IMP-MESE-CORR-S PIC 9(9).
WORKING-STORAGE SECTION.
01  DATA-W PIC 99/99/99.
01  DESCR-FIL-W PIC X(15) VALUE SPACES.
01  C-FIL-PRES PIC XX VALUE "NO".
    88 FIL-PRES VALUE "SI".
01  EOF-W PIC XX VALUE "NO".
     88 EOF VALUE "SI".
01  EOF-T-W PIC XX VALUE "NO".
     88 EOF-T VALUE "SI".
01  TAB-FILIALI.
    02 ELEM-TAB OCCURS 13 TIMES
                INDEXED BY IND.
        03  COD-FIL-T PIC X(4).
        03  DESCR-FIL-T PIC X(15).
        03  IMP-TOT-T PIC 9(12).
        03  IMP-MESE-CORR-T PIC 9(9).
77  IMP-ALTRE-FIL PIC 9(9) VALUE ZERO.
```

```
REPORT SECTION.
RD  TABUL-CC
    CONTROLS ARE FINAL COD-FIL COD-CL      PAGE LIMIT 66 LINES
    HEADING 5   FIRST DETAIL 12   LAST DETAIL 45   FOOTING 55.
01  TYPE IS RH NEXT GROUP NEXT PAGE.
    02 LINE 30 COLUMN 30 PIC X(25)
       VALUE "TABULATO CONTI CORRENTI".
    02 LINE 32 COLUMN 35 PIC X(12)
       VALUE "PER FILIALI".
    02 LINE 34.
        03 COLUMN 35 PIC XX VALUE "AL".
        03 COLUMN 41 PIC X(8) SOURCE DATA-W.
01  INT-P TYPE IS PH.
    02 LINE 5.
        03 COLUMN 50 PIC X(7) VALUE "PAG.N. ".
        03 COLUMN 60 PIC Z(5)9 SOURCE PAGE-COUNTER.
01  RIGA-ST TYPE IS DE LINE PLUS 2.
    02 COLUMN 18 PIC X(8) SOURCE DATA-M .
    02 COLUMN 30 PIC X(10) SOURCE DESCR.
    02 COLUMN 44 PIC Z.ZZZ.ZZ9 SOURCE IMP.
01  TYPE IS PF LINE 61.
    02 COLUMN 60 PIC XXX VALUE "-*-".
01  CH-CL  TYPE IS CH COD-CL.
    02 LINE PLUS 3 COLUMN 5 PIC X(40) SOURCE ANAG.
    02 LINE PLUS 3 COLUMN 20 PIC X(33)
       VALUE "DATA       MOVIMENTO       IMPORTO".
01  CH-FIL  TYPE IS CH COD-FIL LINE 13.
    02 COLUMN 35 PIC X(7) VALUE "FILIALE".
    02 COLUMN 47 PIC X(4) SOURCE COD-FIL.
    02 COLUMN 52 PIC X VALUE ":".
    02 COLUMN 57 PIC X(15) SOURCE DESCR-FIL-W.
01  CF-CL TYPE IS CF COD-CL LINE PLUS 4
         NEXT GROUP NEXT PAGE.
    02 COLUMN 55 PIC X(4) VALUE "TOT.".
    02 DEP-TOT-CL COLUMN 61 PIC ZZ.ZZZ.ZZ9  SUM IMP.
01  CF-FIL TYPE IS CF COD-FIL LINE 55.
    02 COLUMN 50 PIC X(13) VALUE "TOT. FILIALE ".
    02 DEP-TOT-FIL COLUMN 65 PIC ZZZ.ZZZ.ZZ9 SUM DEP-TOT-CL.
01  CF-FINAL TYPE IS CF FINAL LINE 25.
    02 COLUMN 50 PIC X(13) VALUE "TOT. GENERALE".
    02 COLUMN 67 PIC Z.ZZZ.ZZZ.ZZ9 SUM DEP-TOT-FIL.
RD  TABUL-FIL
    CONTROL IS FINAL     PAGE LIMIT 66 LINES
    HEADING 5    FIRST DETAIL 17    LAST DETAIL 45    FOOTING 55.
01  TYPE IS PH.
    02 LINE 5.
        03 COLUMN 50 PIC X(7) VALUE "PAG.N. ".
        03 COLUMN 60 PIC Z(5)9 SOURCE PAGE-COUNTER OF TABUL-FIL.
    02 LINE 11 COLUMN 40 PIC X(20) VALUE "SITUAZIONE FILIALI".
    02 LINE 15 COLUMN 20 PIC X(59)
     VALUE "COD-FIL   DESCR-FILIALE          IMPORTO TOT    IMP MESE CORR".
01  RIGA-ST-TAB TYPE IS DE LINE PLUS 2.
    02 COLUMN 20 PIC X(4) SOURCE COD-FIL-S.
    02 COLUMN 30 PIC X(15) SOURCE DESCR-FIL-S.
    02 COLUMN 48 PIC ZZZ.ZZZ.ZZZ.ZZ9 SOURCE IMP-TOT-S.
    02 COLUMN 68 PIC ZZZ.ZZZ.ZZ9 SOURCE IMP-MESE-CORR-S.
01  TYPE CF FINAL LINE PLUS 6.
    02 COLUMN 30 PIC X(22) VALUE "IMPORTO ALTRE FILIALI".
    02 COLUMN 68 PIC ZZZ.ZZZ.ZZ9 SOURCE IMP-ALTRE-FIL.
01  TYPE IS PF LINE 61.
    02 COLUMN 60 PIC XXX VALUE "-*-".
01  REP-F TYPE IS RF .
    02 LINE 65  COLUMN 30 PIC X VALUE SPACES.
```

```
PROCEDURE DIVISION.
DECLARATIVES.
RICERCA-TAB SECTION.
    USE BEFORE REPORTING CH-FIL.
RIC-TAB.
    SET IND TO 1
    SEARCH ELEM-TAB
           AT END   MOVE SPACES TO DESCR-FIL-W
           WHEN COD-FIL-T(IND) = COD-FIL
                MOVE "SI" TO C-FIL-PRES
                MOVE DESCR-FIL-T(IND) TO DESCR-FIL-W.
FI-TR-FIL SECTION.
    USE BEFORE REPORTING CF-FIL.
FINE-FIL.
    IF FIL-PRES  MOVE "NO" TO C-FIL-PRES
                 ADD IMP-MESE-CORR-T(IND) TO IMP-TOT-T(IND)
                 MOVE ZERO TO IMP-MESE-CORR-T(IND)
                 ADD DEP-TOT-FIL TO IMP-MESE-CORR-T(IND)
    ELSE  ADD DEP-TOT-FIL TO IMP-ALTRE-FIL.
END DECLARATIVES.
MAIN SECTION.
ALBERO.
*
*
**************    ALBERO DEL PROGRAMMA    **************
*
*    IN-PROG  :
          PERFORM IN-STANDARD.
          PERFORM CARICA-TAB UNTIL EOF-T.
*    TR-INTEST    (RW)
*    TR-DATI-FILIALI  :
*         IN-TRATT
*         TR-FIL UNTIL EOF  :
*              IN-TR-FIL  :
*                   IN-FIL-STANDARD    (RW)
*                   RICERCA-TAB    (USE)
*              TR-INTEST-FIL    (RW)
*              TR-CLI UNTIL EOF OR COD-FIL NOT =  :
*                   IN-TR-CLI    (RW)
*                   TR-INTEST-CLI    (RW)
                    PERFORM TR-MOV UNTIL EOF.
*                                       OR COD-FIL NOT =
*                                       OR COD-CLI NOT =
*                   TR-TOT-CLI    (RW)
*                   FI-TR-CLI
*              TR-TOT-FIL    (RW)
*              FI-TR-FIL    (USE)
*         TR-TOT-GEN    (RW)
          PERFORM FI-TRATT.
*    TR-TAB-FIL  :
          SORT FL-SORT ON ASCENDING KEY IMP-TOT-S
               WITH DUPLICATES IN SEQUENCE
               INPUT PROCEDURE IS IN-TR-TAB-STAND THRU PREP-SORT
               OUTPUT PROCEDURE IS TR-INT THRU FI-TR-TAB.
*    FI-PROG
     STOP RUN.
*
*
```

```
**************       MODULI OPERATIVI       ***************
*
 IN-STANDARD SECTION.
 INIZIO.
      OPEN INPUT FL-INP FL-TAB
      OPEN OUTPUT FL-STAMPA.
      ACCEPT DATA-W FROM DATE
      INITIATE TABUL-CC
      READ FL-INP AT END MOVE "SI" TO EOF-W.
      SET IND TO 1.
      READ FL-TAB AT END MOVE "SI" TO EOF-T-W.
*
*
 CARICA-TAB SECTION.
 CARICAMENTO.
      MOVE REC-TAB TO ELEM-TAB(IND)
      SET IND UP BY 1
      READ FL-TAB AT END MOVE "SI" TO EOF-T-W.
*
*
 TR-MOV SECTION.
 TRATTA-MOV.
      IF TSK = "M" GENERATE RIGA-ST.
      READ FL-INP AT END MOVE "SI" TO EOF-W.
*
*
 FI-TRATT SECTION.
 FINE-TRATT.
      TERMINATE TABUL-CC.
*
*
 IN-TR-TAB-STAND SECTION.
 IN-TAB-STAND.
      INITIATE TABUL-FIL
      ADD 1 TO PAGE-COUNTER OF TABUL-CC
      MOVE PAGE-COUNTER OF TABUL-CC TO PAGE-COUNTER OF TABUL-FIL.
      SET IND TO 1.
*
*
 PREP-SORT SECTION.
 PREP.
      RELEASE REC-SORT FROM ELEM-TAB(IND)
      SET IND UP BY 1
      IF IND NOT > 11 GO TO PREP.
*
*
 TR-INT SECTION.
 TRATTA-INT.
      SET IND TO 1.
      RETURN FL-SORT RECORD AT END GO TO FI-TR-TAB.
*
*
 TR-RIGA-TAB SECTION.
 TRATTA-RIGA.
      GENERATE RIGA-ST-TAB
      RETURN FL-SORT RECORD AT END GO TO FI-TR-TAB.
      GO TO TRATTA-RIGA.
*
*
*"TR-IMP-RESID"  (RW)
*
*
 FI-TR-TAB SECTION.
 FI-TR-TABELLA.
      TERMINATE TABUL-FIL
      CLOSE FL-INP FL-TAB FL-STAMPA.
```

PROGETTAZIONE E REALIZZAZIONE DI APPLICAZIONI IN AMBIENTE TP

T. Bettinazzi(°) - A. Maserati(°) - F. Monzani(°) - E. Spoletini(+*) - E. Toscani(+)

Riassunto

Scopo di questa nota è illustrare alcuni aspetti metodologici riguardanti la progettazione di applicazioni, in ambiente TP con monitor, soprattutto per quanto riguarda la fase di disegno.
Il punto di vista è quello di colui che vede il Sistema TP come un oggetto che offre una serie di funzionalità e un insieme di limiti logico-applicativi e a cui necessitano opportuni strumenti concettuali e linguistici che lo supportino in ogni fase del lavoro.
Detti strumenti devono mettere l'utente in grado di affrontare con atteggiamento sistemistico il problema della realizzazione delle funzioni previste tenendo conto di tutti gli aspetti che questa attività comporta.

Abstract

The aim of this paper is to show some methodological aspects concerning the applications project in TP-environment with monitor, mostly in that concerning the design phase.
The point of view is that of a person who examines the TP-System as an object that gives a functionalities series and a set of logical-applicational limits and to whom needs appropriate tools (conceptual and linguistic) that support him in any work phase.
These tools must enable the user to tackle, with systematic attitude, the problem of foreseeing functions realization, considering any aspects that this activity requires.

BATCH

Il batch è un ambiente in cui una attività per poter essere eseguita deve aver allocate le risorse utili alla sua esecuzione.

TP (ottica utente)

TP è un ambiente in cui una transazione viene eseguita a prescindere dalle risorse disponibili: l'allocazione delle risorse avviene durante l'esecuzione. Le elaborazioni sono in concorrenza su risorse comuni.

RISORSE IN AMBIENTE BATCH

In un ambiente batch si distinguono due tipi di risorse:

— quelle strettamente legate al sistema:
- sistema operativo batch-oriented
- utilities
- memoria centrale
- memorie di massa

— e le procedure applicative.

RISORSE IN AMBIENTE TP

In un ambiente TP si distinguono due tipi di risorse:

— quelle strettamente legate al sistema:
- sistema operativo di tipo:
  + batch con sistema per la gestione TP
    oppure
  + TP
- utilities, memorie, linee, stazioni terminali

— e le procedure applicative batch e TP.

TIPO D'USO IN AMBIENTE BATCH

- 1 solo utente per volta per una procedura
- 1 sola esecuzione per volta per una procedura

TIPO D'USO IN AMBIENTE TP

- n utenti per una procedura
- n esecuzioni in concorrenza per una procedura

(°) ATROPO srl
(+) HISI Direzione Formazione
(*) Univ. di Milano — Ist. di Matematica

TRANSAZIONE

Transazione: rapporto in cui due enti si scambiano informazioni per raggiungere un obiettivo

È l'attività fatta per eseguire delle funzioni definite dall'utente; è caratterizzata dai punti:

di START: il sistema riceve un messaggio
di STOP: le funzioni richieste dall'utente sono eseguite.

TEMPO REALE

Tempo Reale: in ottica utente è un ambiente in cui una attività deve essere eseguita in condizioni di tempo limitate.

PROBLEMATICHE DELL'AMBIENTE TP

- costruzione delle applicazioni: gestione transazioni concorrenti
  concorrenza uso dati e memoria
  rapporto con l'utente
  security
  integrity
- gestione dei problemi riguardanti la trasmissione dei dati:
  gestione linea
  gestione errori di trasmissione
  gestione dialogo sistema/stazione
  ripristino e riavviamento.

PRIMI APPROCCI AL PROBLEMA

- Invenzione di linguaggi orientati alla soluzione di problemi TP.
- Costruzione di sistemi Software che permettano all'utente di scambiare "messaggi" tramite una linea.

LINEE DI TENDENZA NELL'EVOLUZIONE DEI SISTEMI

- Centralizzare la gestione dei problemi connessi alla trasmissione dati lasciando all'utente solo il problema della costruzione delle applicazioni (transazioni).

LA SOLUZIONE

Costruzione di sistemi per la gestione

- dell'ambiente TP: esecuzione delle transazioni
  concorrenza (esecuzione e risorse)
  dialogo sistema/terminali
  security - ripristino - riavviamento
  gestione messaggi
- delle linee di trasmissione.

EFFETTI SUI SISTEMI

- prima fase: espansione dei sistemi operativi con l'implementazione di sottosistemi a cui delegare la gestione delle problematiche TP.
- seconda fase: evoluzione dei sistemi operativi integrando le problematiche TP e batch in un'unica soluzione di trattamento.

REALTA' DIFFERENTI

- Sistemi che supportano il TP solo con un linguaggio TP-oriented
  consentono la stesura di programmi TP senza nessun supporto
- Sistemi che supportano il TP con monitors TP che integrano il sistema operativo batch-oriented
  consentono la stesura di programmi TP col supporto di monitors TP per i problemi di trasmissione dati
- Sistemi che gestiscono il TP in modo analogo al batch supportandolo integralmente
  consentono la stesura di programmi TP in modo assolutamente analogo al batch.

PHOS-TP

PHOS-TP: è una applicazione della filosofia di lavoro PHOS
è quindi una
metodologia di definizione
e di progettazione
di applicazioni
in ambiente TP

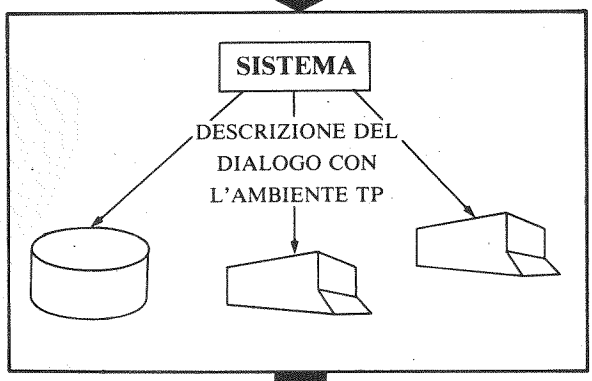
PHOS-TP: obiettivo

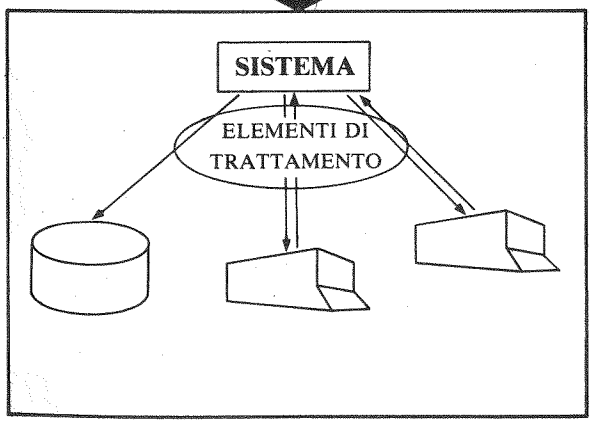
ANALISI DISEGNO SVILUPPO MANUT.

rende sistematico il processo di costruzione di software a fronte di problematiche TP.

PHOS-TP: caratteristiche

- linee guida precise per la costruzione delle transazioni a partire dalle specifiche
- supporti concettuali
- linguaggio grafico
- tecnica di sviluppo top-down
- adozione coerente delle strutture di controllo della programmazione strutturata
- indipendenza dall'ambiente di programmazione

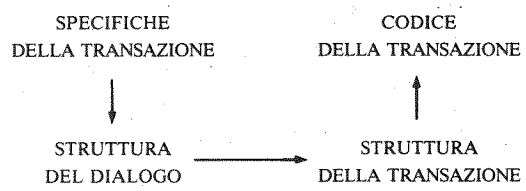




DIALOGO

È L'INSIEME
DI MESSAGGI E DI DATI
CHE SISTEMA E TERMINALI
SI SCAMBIANO ALLO SCOPO
DI RAGGIUNGERE
UN CERTO OBIETTIVO

PHOS-TP
logica

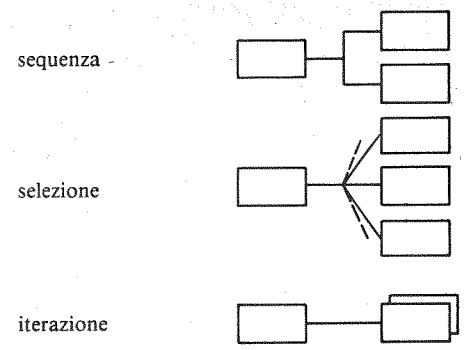


- linguaggio di descrizione del dialogo e della transazione
- funzione di trasferimento tra due realtà descritte

STRUTTURA DEL DIALOGO

Modo di descrivere il dialogo secondo regole di

- definizione degli elementi
- associazione



UNITA' LOGICA DEL DIALOGO (ULD)

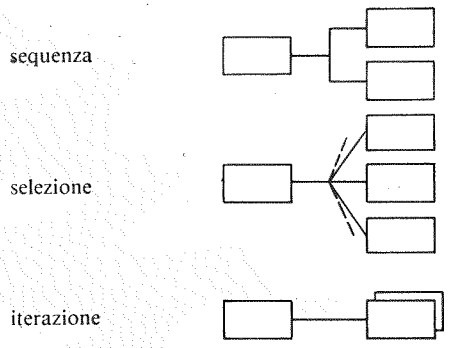
insieme delle informazioni
che definiscono
un soggetto
di trattamento

STRUTTURA DELLA TRANSAZIONE

È il progetto dello strumento che consente di raggiungere gli obiettivi definiti dalle specifiche della transazione.

È la descrizione dell'organizzazione dei trattamenti che espletano la funzione richiesta secondo regole di

- definizione degli elementi
- associazione



UNITA' DI TRATTAMENTO

Insieme delle istruzioni necessarie ad esaurire il trattamento di una unità logica del dialogo

- UNITA' LOGICA DEL DIALOGO → SOGGETTO DI TRATTAMENTO
- UNITA' DI TRATTAMENTO DEL DIALOGO → SOGGETTO DI TRATTAMENTO

PARALLELO BATCH	TP
UL → OUT e INP	ULD → STRUTTURA DEL DIALOGO
UT → PROG	UTD → STRUTTURA DELLA TRANSAZIONE
UL → SOGGETTO DI	ULD → SOGGETTO DI
UT → TRATTAMENTO	UTD → TRATTAMENTO

U = unità, L = logica, T = trattamento, D = dialogo

FASI DELLA METODOLOGIA PHOS-TP

- A partire dalle specifiche del problema individuare le Unità Logiche del Dialogo

⇩

- Procedere nel processo di raffinamento top-down fino ad arrivare all'individuazione delle Unità Logiche elementari

⇩

- Individuare i records logici di output e i messaggi di I/O

⇩

- Ricavare la STRUTTURA DELLA TRANSAZIONE a partire dalla STRUTTURA DEL DIALOGO, facendo corrispondere una UTD ad ogni ULD

⇩

- Inserire blocchi di inizio e fine trattamento

⇩

- inserire sulla struttura della transazione le istruzioni di interazione fra sistema e stazioni terminali:
  — RECEIVE
  — SEND

⇩

- Individuare le routines percorrendo la struttura della transazione ed escludendo in ciascuna routine tutta quella parte che sta tra una RECEIVE e la successiva SEND di I/O.
  Questo tipo di segmentazione fornisce la riduzione in routines di dimensione massima possibile rispetto al complesso dello scambio dei messaggi: ulteriori segmentazioni sono sempre applicabili.

E I CONFLITTI ...

Per quanto riguarda le singole routines possono verificarsi situazioni conflittuali "tipiche" di un programma batch..., saranno affrontate con le consuete tecniche messe a disposizione da PHOS.
Siamo invece in assenza di conflitti per quanto riguarda lo scambio di messaggi in relazione al dialogo in quanto l'input è richiesto tramite maschere in modo conforme alle aspettative del sistema.

CON MONITOR ...E SENZA

In questa nota abbiamo dedicato spazio al problema della progettazione di applicazioni in TP facendo più volte riferimento al caso di lavoro in presenza di monitor, non perché le modalità di trattamento siano diverse o particolari, ma perché l'utenza di questo Software è troppo spesso allontanata dalla realizzazione di applicazioni in modo conforme alla logica del problema da presunti condizionamenti e vincoli dell'ambiente software. Dette limitazioni d'ambiente esistono, ma non sono certamente al più alto livello gerarchico, il prossimo esempio, realizzato in ambiente TPS Livello 62, mostra come le limitazioni imposte dall'ambiente intervengano praticamente solo in fase implementativa.

UN ESEMPIO:
Gestione archivio clienti

Scopo della transazione proposta è di realizzare l'aggiornamento on line di un archivio clienti.

La transazione ha inizio con l'invio di una maschera di richiesta codice.

Possono verificarsi tre casi:

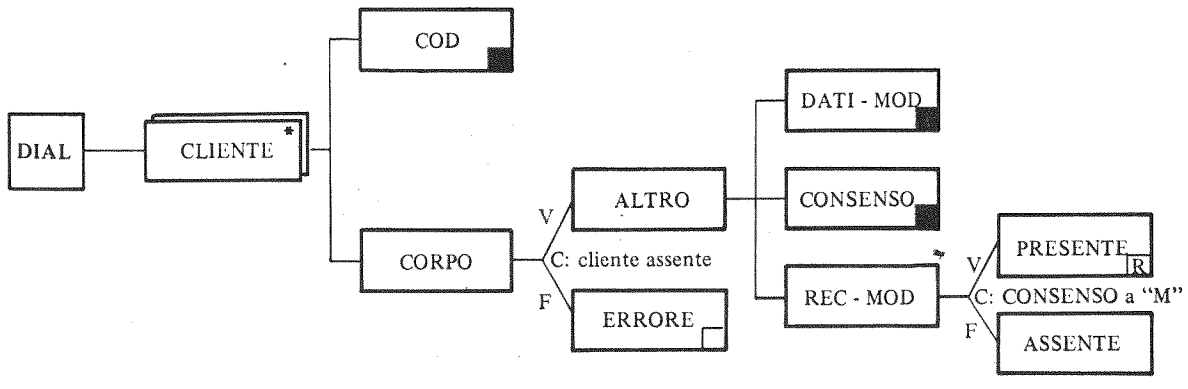
1 - record archivio presente

- vengono inviati su video i dati contenuti in archivio per essere modificati dal terminalista
- viene quindi proposta una maschera di richiesta consenso alla modifica: l'aggiornamento del file avviene solo in caso di risposta affermativa.

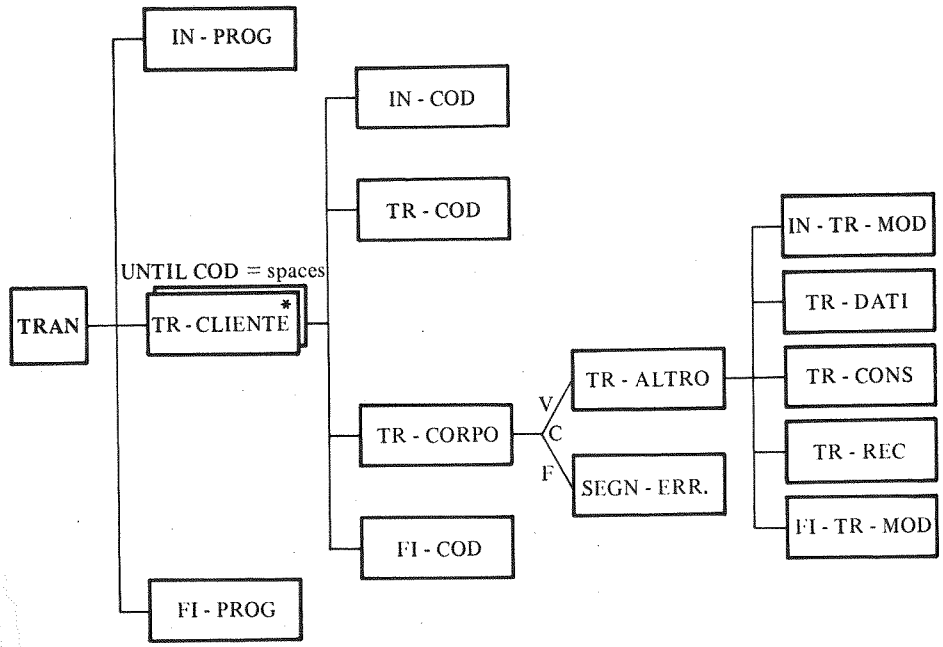
2 - record assente

- l'assenza viene segnalata con l'invio su video della costante "non presente".

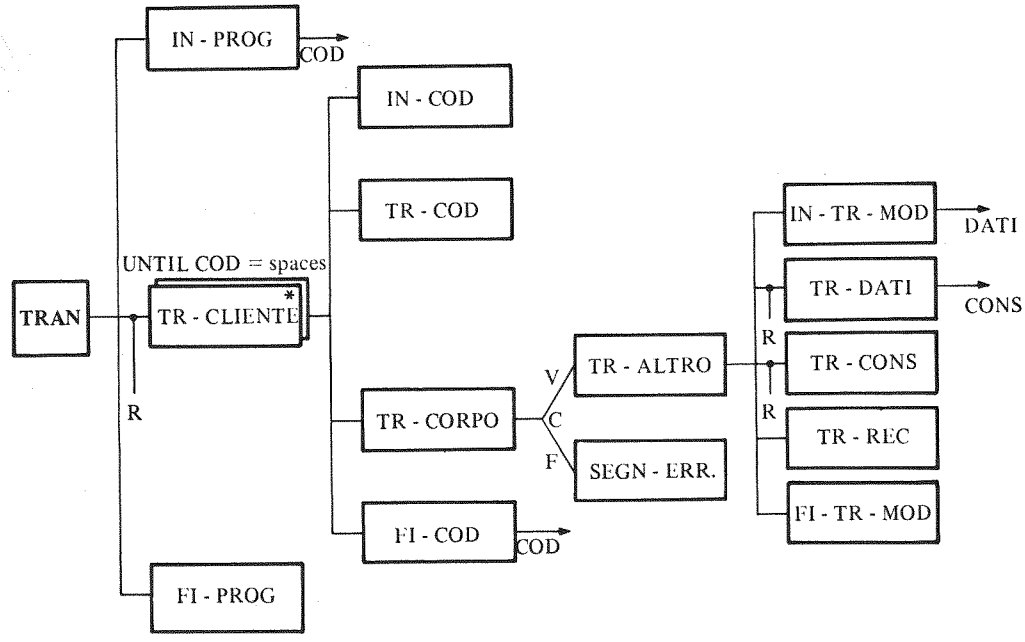
Al termine del trattamento viene riproposta la maschera di richiesta codice e la transazione termina inserendo un valore nullo.



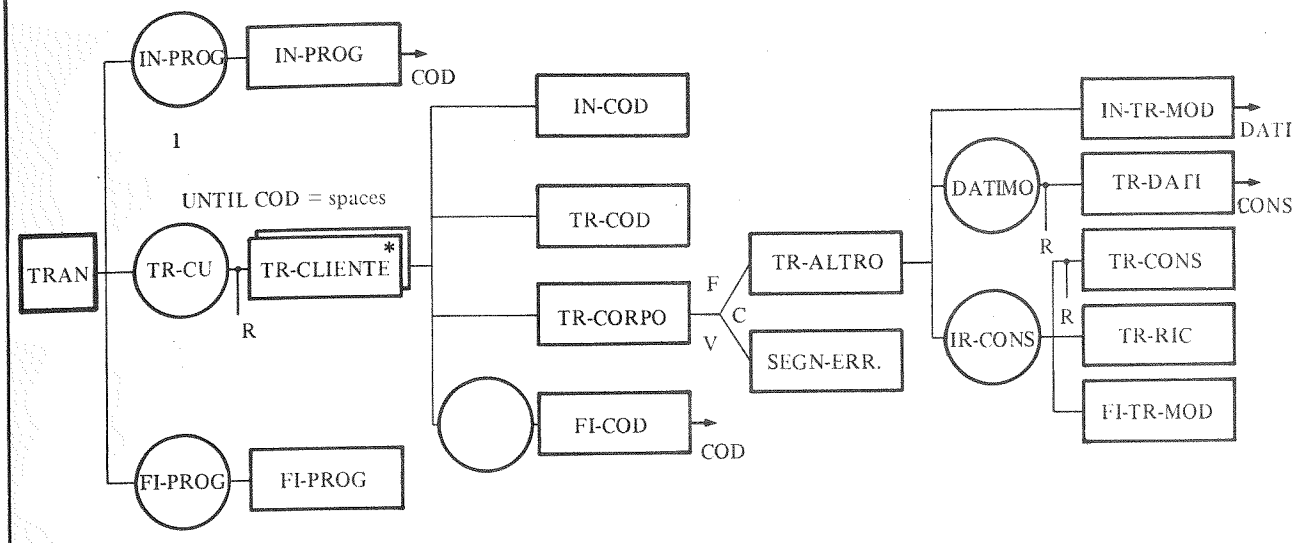
Il dialogo presenta una struttura iterativa a livello "CLIENTE". Le informazioni che esaminiamo e definiscono questo soggetto sono descritte attraverso un procedimento top-down evidenziando, attraverso un simbolismo grafico opportuno, le unità logiche corrispondenti rispettivamente a: maschere di I/O (COD, DATI-MOD, CONSENSO), maschere di solo output (ERRORE), risultati definitivi (REC-MOD).



La struttura della transazione è direttamente dedotta dalla struttura del dialogo associando ad ogni unità logica di trattamento ed attribuito ad ogni livello gerarchico un modulo di inizio e fine.



La struttura della transazione è completata con l'inserimento delle istruzioni di input/output da terminale. Questa allocazione è realizzata in modo automatico sulla base della posizione, nella struttura del dialogo, dei simboli identificativi delle maschere di I/O. Oltre a queste possono essere evidenziate, in analogia al caso batch, anche tutte le operazioni di Input/Output su files come read, write, delete, etc.



La individuazione, sulla struttura della transazione, delle unità di trattamento relative alle diverse TPR è realizzata in base alla posizione delle istruzioni Send/Receive. L'inizio di una TPR è identificato dal simbolo "O" ed il suo contenuto dalla porzione di struttura racchiusa tra due delimitatori consecutivi.

CODIFICA di INPROG e maschera COD

```
010 IDENTIFICATION DIVISION.
020 PROGRAM-ID.  INPROG.
030*
040 DATA DIVISION.
050 WORKING-STORAGE.
060 01   DAT.
070      02  AA           PIC 99.
080      02  MM           PIC 99.
090      02  GG           PIC 99.
100 01   DAT-MOD.
110      02  GG-M         PIC 99.
120      02  MM-M         PIC 99.
130      02  AA-M         PIC 99.
         TERMINAL DATA IS TERM-DAT.
     01  TERM-DAT.
         02  FILLER       PIC X(41).
         02  DAT-OUT      PIC 99/99/99.
150*
160*****************************************************************
010 PROCEDURE DIVISION.
020 INIZIO.
030      ACCEPT DAT FROM DATE
040      MOVE AA TO AA-M
050      MOVE MM TO MM-M
060      MOVE GG TO GG-M
070      MOVE DAT-MOD TO DAT-OUT     .
080%     SEND FORM "COD" TO MR "TRCLI" USING DAT-OUT.
090%     ENDMR.
```

```
DIALOGUE-DESCRIPTION COD     FM
DATA-DEFINITION
COD
         IN31  038   1 22
                                  "*** INTERROGAZIONE E AGGIORNAMENTO ***"
         INT2  0 8   1 71    1
         INT3  016   2 32
                                  "FLUSSO ARTICOLI"
         INT4  080   3  1
                                 *"-"
         CODI  017   5  6
                                  "CODICE ARTICOLO :"
         VALCODI 6   5 25         1
         DELCODO 1   5 31
                                  "*"
DIALOGUE-DEFINITION
   CLR  SCREEN
   INS  COD
END-DIALOGUE
```

CODIFICA di TRCLI e maschera DATI

```
010 IDENTIFICATION DIVISION.
020 PROGRAM-ID. TRCLI.
030 DATA DIVISION.
040 INPUT-MESSAGE.
050      02  CODICE-CLI   PIC X(6).
060 WORKING-STORAGE.
070 01   MESSAGGIO-OUT    PIC X(12).
080 01   DATI-CLIENTE.
120      02  DES-OUT      PIC X(20).
090      02  QTA-OUT      PIC 9(6).
100      02  UM-OUT       PIC XX.
110      02  PRU-OUT      PIC 9(7).
130*
140      TERMINAL DATA IS TERM-DAT.
150 01   TERM-DAT.
160      02  DEP-COD      PIC X(6).
170      02  FILLER       PIC X(35).
         02  DAT-OUT      PIC 99/99/99.
180*
190*****************************************************************
200*
010 PROCEDURE DIVISION.
010*
011 CONTROLLO-ITER.
012      IF CODICE-CLI = SPACE GO TO FI-PROG.
020 IN-TR-CLI.
030      MOVE CODICE-CLI TO COD-ART DEP-COD.
040%      READ ARKCLI INVALID KEY SEG-ERR ON LOCK SEG-STOP
050*TR-COD.  (VUOTO)
060 IN-CORPO.
070      MOVE QTA-ART TO QTA-OUT
080      MOVE UM-ART  TO UM-OUT
090      MOVE PRU-ART TO PRU-OUT
100      MOVE DES-ART TO DES-OUT
110%     SEND FORM "DATI" TO MR "DATIMO" USING DATI-CLIENTE.
120%     ENDMR.
130*
140 SEG-ERR.
170*          ** LA MR "FICLI" DEVE SOLTANTO INVIARE LA MASCHERA
180*          ** DI RICHIESTA CODICE : PUO' ESSERE QUINDI OMESSA
190*          ** INSERENDO TALE ISTRUZIONE IN SEG-ERR E FI-CORPO
150      MOVE "NON PRESENTE" TO MESSAGGIO-OUT
155      DISPLAY MESSAGGIO-OUT.
160%     SEND FORM "ERR" USING MESSAGGIO-OUT.
   %     SEND FORM "COD" TO MR "TRCLI" USING DAT-OUT.
201%     ENDTR.
202%     ENDMR.
020 SEG-STOP.
030      MOVE "BLOCCATO" TO MESSAGGIO-OUT
040%     SEND FORM "ERR" USING MESSAGGIO-OUT.
050%     READ ARKCLI INVALID KEY SEG-ERR.
060      GO TO IN-CORPO.
070*
080 FI-PROG.
090*          ** LA MR "FI-PROG" DOVREBBE CONTENERE SOLO LA FINE ITP:
100*          ** TALE ISTRUZIONE VIENE INSERITA DIRETTAMENTE IN QUESTA
110*          ** MR
120%     ENDITP.
```

```
DIALOGUE-DESCRIPTION DATI    FM
DATA-DEFINITION
DATI
         C1     013    8 10
                                  "DESCRIZIONE :"
         I1     U20    8 25    1   1
         DEL1   0 1    8 45
                                  "*"
         C2     013    9 10
                                  "QUANTITA'    :"
         I2     U 6    9 25   21  21
         DEL2   0 1    9 31
                                  "*"
         C3     013   10 10
                                  "UN.MISURA    :"
         I3     U 2   10 25   27  27
         DEL3   0 1   10 27
                                  "*"
         C4     013   11 10
                                  "PREZZO UNITA:"
         I4     U 7   11 25   29  29
         DEL4   0 1   11 32
                                  "*"
DIALOGUE-DEFINITION
   INS  DATI
END-DIALOGUE
```

CODIFICA di DATIMO e maschera CONS

```
010 IDENTIFICATION DIVISION.
020 PROGRAM-ID. DATIMO.
030 DATA DIVISION.
040 INPUT-MESSAGE.
050     02  DATI-MOD.
090         03  DES-MOD PIC X(20).
060         03  QTA-MOD PIC 9(6).
070         03  UM-MOD  PIC XX.
080         03  PRU-MOD PIC 9(7).
100     TERMINAL DATA IS TERM-DAT.
110 01  TERM-DAT.
120     02  FILLER      PIC X(6).
130     02  DEP-DATI    PIC X(35).
140*
150*************************************************
160*
170 PROCEDURE DIVISION.
180 DATI-MODI.
010     MOVE DATI-MOD TO DEP-DATI
020%    SEND FORM "CONS" TO MR "TRCON" .
030%    ENDMR.
```

```
DIALOGUE-DESCRIPTION CONS    FM
DATA-DEFINITION
CONS
        SEGN   028  16  2
                                "DIGITARE:
        VAL    I 1  16 43       1     'M' PER MODIFICARE"
        DEL    0 1  16 44
                                "*"
DIALOGUE-DEFINITION
   INS  CONS
END-DIALOGUE
```

CODIFICA di TRCON (che usa maschera COD)

```
010 IDENTIFICATION DIVISION.
020 PROGRAM-ID.  TRCON.
030 DATA DIVISION.
040 INPUT-MESSAGE.
050     02  CONS        PIC X.
055 WORKING-STORAGE SECTION.
060     TERMINAL DATA IS TERM-DAT.
070 01  TERM-DAT.
080     02  DEP-COD     PIC X(6).
090     02  DEP-DATI.
130         03  DEP-DES PIC X(20).
100         03  DEP-QTA PIC 9(6).
110         03  DEP-UM  PIC XX.
120         03  DEP-PRU PIC 9(7).
        02  DAT-OUT     PIC 99/99/99.
140*
150*************************************************
160*
170 PROCEDURE DIVISION.
180 TR-CONS-MOD.
010     IF CONS NOT = "M" GO TO FI-CORPO.
020     MOVE DEP-COD TO COD-ART
030%    READ ARKCLI INVALID KEY SEGN-ERR.
040     MOVE DEP-QTA TO QTA-ART
050     MOVE DEP-UM  TO UM-ART
060     MOVE DEP-PRU TO PRU-ART
070     MOVE DEP-DES TO DES-ART
080%    REWRITE ARKCLI.
090*
100 FI-CORPO.
   %    SEND FORM "COD" TO MR "TRCLI" USING DAT-OUT.
120%    ENDTR.
130%    ENDMR.
140*
150 SEGN-ERR.
160     DISPLAY "ERRORE IN ARK" COD-ART
170%    ENDITP.
```

RIFERIMENTI

[1] C. Bettinazzi, A. Maserati, F. Monzani, E. Spoletini, E. Toscani, UN APPROCCIO SISTEMATICO PER IL DISEGNO DI APPLICAZIONI TP, presentato alla Rivista di Informatica

[2] E Spoletini, REAL-TIME APPLICATIONS AND PRACTICAL TOOLS FOR PROGRAMMING IN THE PHOS ENVIRONMENT, Proc. V Annual Honeywell International Software Conference, Bloomington, marzo 1981

Per altri riferimenti, vedi in: G. Magnoni, E. Spoletini, E. Toscani, NOTE SULL'USO DI FUNZIONALITÀ COBOL IN AMBITO PHOS, NOTE DI SOFTWARE 15/16

THE RIGHT WAY TO COMMUNICATE ACROSS DISTRIBUTED MACHINES

Piercarlo Grandi
Università di Milano, Istituto di Cibernetica

Riassunto

La caratteristica principale della comunicazione fra processi su macchine diverse è che essa può fallire, e che perciò essa deve sempre consistere di baratti di informazioni, così che ciascun processo può assicurarsi dell'esistenza del suo interlocutore. La definizione della semantica degli scambi, logicamente necessari quando non esiste un supervisore centrale che controlla e perciò garantisce la comunicazione, è alquanto diversa da quella dell'invio di messaggi o del passaggio di argomenti, che sono i protocolli usati per comunicazione fra processi sulla stessa macchina. Viene tratteggiata una definizione accettabile delle primitive per il controllo distribuito, e quindi la loro implementazione in un sistema convenzionale. Vengono qui discussi alcuni dettagli interessanti e viene dimostrata la essenzialità ed efficienza del modello del baratto.

Abstract

The main characteristic of communication among processes on different machines is that communication can fail. Therefore communication must always consist of barters of information, so that each process can be aware of the status of its partner. The definition and semantics of barters, which are necessary from a logical viewpoint when no central supervisor exists for monitoring and guaranteeing communication among processes, is quite different from that of message passing or argument passing, which are the protocols used among processes on the same machine. An acceptable sketch of definition of distributed communication primitives, and their implementation in a conventional system, is described. Some interesting details are discussed, and the clear minimality and efficiency of barters is demonstrated.

1. MODELS OF PROGRAMMING

The most common model of programming is the procedural, in which programs are procedures composed of procedures, down to a set of primitive procedures whose properties are guaranteed by the correctness of their interpreter (microcode or circuitry), that is the CPU. Tipical computer systems to which this model applies are large mainframe timeshared computers in which the interpreter allows for parallelism and nondeterminism in the execution of procedures. This conceptual model has been described and analyzed in many papers, among which the works of Prof. Dijkstra. Well designed hardware and software systems efficiently implement the possibility of such decomposition.

1.1. Distributed computing

Efficient small machines have become available at low cost, thus threatening th cost advantages of big centralized computers. Thus the problem of devising a model in which program decomposition is such that parts of the same program may be executed on different machines has become expecially relevant.

The two conceptual problems are that whilst in the single machine environment communications among parts of a program may take place simply by passing aroung pointers to the objects to communicate (because the same memory is used by everyone), this is no longer true in distributed systems; and that while in a single machine there is a single scheduler that manages all processes and guarantees they can synchronize, distributed machines have distinct and uncooperating schedulers and synchronization among them is managed ad hoc and with many possible pitfalls.

1.2. Two orthogonal problems

As is immediately apparent the two above problems are completely different; it is possible to imagine an environment in which one is present and the other is not.

For example, there are some machines in which programs do not have common memory but where all of them are controlled by the same scheduler. And there are systems (even if much less common,

Acknowledgments

The author gratefully acknowledges the continuing support and the lively discussions with Prof. Degli Antoni. He thanks also Paolo Rossi for having read drafts of this paper, and Virginia Bacchini for having usefully commented this paper, and R. Polillo.
This research has been developed in the frame of the CP-Project, joint research project between University of Milano, Istituto di Cibernetica, and Honeywell I.S.I.

e.g. the global memory interconnected Burroughs 6700s) where there is global memory but no central scheduler.

The two problems are: how to pass information across distinct machines, when their schedulers cooperate, and how to cope with the case of uncooperating schedulers.

Some so called solutions have been devised for the first problem, while the second has almost been ignored. The disturbing aspect is the claim made by so many of the 'solutors' of the first problem that their ideas apply in general also to real life distributed systems. Unfortunately this is not the case, because the dominant problem of distributed systems is the second, that is the problem of either make schedulers cooperate or to devise the correct approach to allow programs under uncooperating schedulers to communicate and synchronize.

2. THE GLOBAL MEMORY

The first problem (passing information around across machines) does not arise if the machines cooperate under a single scheduler, in the sense that is possible to make invisible to any program the fact that part of the data being used is resident in another machine.

It has been however proposed not to hide the actual location of objects, but to avoid the problem entirely passing around copies of the objects and not just pointers to them. It can thus be suggested to give to any process that wants to manipulate it a copy of the object: when one process modifies its copy, it modifies all the others. This is quite silly, possibly resulting in frustration of most process attempts to deal with an object before it becomes obsolete, and also because one can use some kind of critical region to protect the current 'original' copy of the object, and define two kinds of requests for copying an object (one for looking at, the other for altering it), and an 'end request signal'. The 'get copy for altering' request is satisfied only when every other known holder of a copy has signaled it is through with it, and the 'get copy for looking' request only when no one has a copy for altering it. This obviously assumes the existence of cooperating schedulers.
The reader will have realized that this is the classical solution to the readers and writers problem, with the only difference that any 'reference to an object' has been replaced by 'a copy of the object'.

2.1. Distribution is transparent under a single scheduler

Each processor is attached to a memory box, and processors are attached to other processors. A processor may address its memory box directly, but not the other processor boxes; it may only fetch or store the contents of its own box.

A fundamental property of any object is its distinguishability from any other object; it must have therefore a unique id. Usually such a unique id is its address in the memory box in which it resides. It is obvious that distinct objects may have the same address if they are in distinct boxes. But a program using an object and wanting to synchronize with other programs that operate on the very same object must denote it by an id unique across all machines on which the other programs execute, even if it resides in another's processor memory box. This means that when one process synchronizes with another over some object, the two processes know in which machine the object exists, and use that machine id to distinguish objects in different machines (it is essential that machines be uniquely identified).

2.1.1. The global addressing across memory boxes

Thus the address or the unique id of an object consists of the concatenation of its (unique) machine id and of its (unique) position in the memory box of that machine. But this implies that a processor may distinguish the case in which the object to fetch or to store exists in its own machine (in which case it uses its memory box to do the operation) and the other one, in which it asks its colleague, the processor of the appropriate machine, to do the operation on its memory box.

In other words, memory boxes may be separated, but there exists an addressing convention by virtue of which all such memory boxes belong to the same (virtual) address space: with some simple cooperation among processors attached to each memory box, it is possibile to give the illusion to any program it can directly reference any object in this common memory space.

2.1.2. Access and synchronization in global memory

Access conflicts arising from concurrent programs may then be dealt with exactly the same rules used in single machine systems (provided there exists a centralized control). It must be emphasized that centralized control systems implementing either message passing or argument passing have exactly the same degree of efficiency.

It has been argued that message passing is possibly better, for distributed computing, than passing arguments, as it is more efficient: argument passing requires the monitoring of each fetch and store order executed by the processor to detect addressing of objects in some other machine's memory box and request a copy of the object. Unfortunately, also message passing requires monitoring of every message sent and received, to detect communication of messages within the same machine to avoid copying them but to simply pass pointers instead. It is obvious that the great majority of communication among processes is between two processes on the same machine. This means that, implementing the global memory as shown, it is better to use the procedural way of decomposing programs irrespective of the locus of execution of the procedures; a procedure may call another procedure anywhere in the cooperating schedulers (centralized control) system.

2.2. Why message passing is not better than procedure calls

It may be argued that, in order to eliminate the need for constant monitoring of memory accesses by a procedure to detect a reference to a remote object, it may be sensible to implement both the procedural and the message passing models of program decomposition, so that the programmer may explicitly use procedure calls for communication on the same machine (thus allowing parameters and global variables to be passed by reference) and message passing to request copying for remote communication.

This position is wrong for two reasons. The first is that the efficiency argument of making the programmer explicitly request copying or not to avoid continuous monitoring is very weak, as monitoring should be done anyhow by the processor to implement bound checking, security violation detection, and to implement virtual memory. The other is that it greatly complicates the life to the programmer, makes the structure of the program dependent on the physical placement of its parts, and creates a lot of fuss about synchronization. Message passing has the nasty property that the copies of objects transmitted as parameters tend to become stale, unless the aforementioned protocol for granting copies is used; but it is much easier to implement once and for all it in the simulation of global memory than to require the user to reimplement a version of the readers & writers scheme for every object type; this is for example what happens with Prof. Hewitt's message passing Actor System, in which a serializer is needed for every object type. The better solution is to simulate a global memory, and use a synchronization algorithm valid for any (uninterpreted) object in memory, and transparent to the user.

2.3. The possible implementation

Many machines implement hardware monitoring of fetches and stores; it is therefore often easier to implement, instead of the message passing scheme, the environment for procedure calls in distributed computing. For example, on a segmented virtual memory machine it is entirely possible to flag a segment so that on each access to the segment an operating system routine is called, whose role is to map a copy of a remote object into that segment, giving to the program the illusion that the object resides in that segment.

3. CENTRALIZED OR DECENTRALIZED CONTROL

The previous discussion shows that there is no reason for preferring message passing when the memory boxes are on different machines, and the schedulers of these machines cooperate in order to simulate a centralized control. The problem to tackle is what to do when schedulers cannot cooperate, a thing that has been seldom considered in its implications.

3.1. The logical consequences of decentralized control

3.1.1. The abstract model of program execution

The fact that the absence of a centralized control makes a difference, becomes evident when considering failures. As a matter of fact, if nothing could fail, there would be no detectable difference

between the two cases.

Let's consider a program that manipulates some objects.This program has been recursively decomposed in subprograms, that communicate by referencing a group of common objects. The concurrency in the execution of these subprograms is immaterial, as long as appropriate interlocks (the readers and writers synchronization scheme) are provided to cope with parallelism. Furthermore, nondeterminism (the other consequence of concurrency) is also immaterial (except for the possibility of deadlocks), if each subprogram is executed only when its stated precondition is true. When a program wants an object to be manipulated by some subprogram, it simply instructs the scheduler to give control to the subprogram and to pass to it a reference to the object (or a copy, or any of the numerous possible views of a parameter), and to give control back to itself when the subprogram has done its job.

3.1.2. Failures under centralized control

Suppose now the subprogram fails for any reason: the central scheduler knows it and gives control back to the calling program with an error flag raised, so that it may realize the failure of the called subprogram. The essential point is that the scheduler knows about the state of all the programs because it manages and interpretes all of them (we may consider the scheduler to be the CPU itself -the instruction a program submits to the scheduler to transfer control to a subprogram is the common 'save current address and jump').

3.1.3. Deadlock under centralized control

Deadlock is the problem of centralized control because under centralized control program execution may be non deterministic, and thus many sequences of execution may be legitimately scheduled, with some of them leading to contradictions between the preconditions of the currently executing programs. The scheduler should thus avoid to schedule the programs in sequences that lead to deadlock, but this is impossible, as the scheduler knows the PRESENT preconditions of all the programs, but cannot predict the FUTURE preconditions of programs, and is thus not able to know which sequences of execution will lead to contradictions.

3.1.4. Failure under decentralized control

Under decentralized control, scheduling is distributed, i.e. each machine has its own scheduler. When a process tells the scheduler it wants to communicate with some program on another machine, it wants that some objects be passed to the other program, give it control, and wait for the manipulated objects to be handed back. The problem is that if the program on the other machine fails, its response never arrives, and the scheduler of the machine on which the caller exists has no way to know of the failure and let the caller know of it. It may also be that the communication link between the two machines is broken, and this situation is (almost) indistinguishable from the other, and has the same consequences.

3.2. Abstraction in decentralized control

All this implies that it is logically inconsistent to decompose the program to be executed on more than one machine by levels of abstraction, in which control passes from one level to the other first via a question (a downward passing of objects) and then via the ensuing answer (an upward passing of the modified objects). The reason is that programs cannot be sure of the arrival of the answer from other programs living in different machines (or of getting a failure ratification from the scheduler) or even of the existence of the other machine. This means that mono-directional communication cannot be used without a centralized control.

It is evident that the decomposition paradigm for decentralized control is not abstraction but aggregation; and that there are well structuredness rules for aggregation as well as for abstraction. Whereas in abstraction the rule is that each level of abstraction has one and only one entry and exit, with aggregation each group has one and only one component that communicates with another given group.

The reasons for this are that, in this way, understanding the decomposition becomes much simpler: the status of each communication depends on only one component of the group, and there is no exponential explosion of complexity of interrelationships.

3.3. The differences between the two control regimes

When control is centralized and the (present) status of all programs is known, it is perfectly sensible to use two monodirectional communications in dis-

tinct phases to pass control and hand it back, being sure of receiving either the correct answer or a failure indicator. Moreover, each communication is sure to terminate, unless there is a deadlock, that is unavoidable anyhow (without advance information, that must be provided in the deadlock avoidance scheme of the maximum claim of Habermann).

When control is decentralized, the scheduler on one machine does not know the (present) status of programs on other machines, so a program that communicates with another machine must always make a bidirectional communication in which objects are exchanged to guarantee both partners that the other is alive. Also, a timeout window must be established by each of the programs to ensure that the exchange operation terminates even if the other program fails to accomplish its half of the exchange (to provide its object).

3.4. Timeout is the problem for decentralized control

The problem of decentralized control is timeout race, and not deadlock. Even if the two programs on the two machines are alive and the communication link between them operational, it may happen that the timeout period for one is sistematically too short, and therefore communication may never succeed. Conversely, deadlock is impossible: if two programs come to a point in their exchanges where each is waiting for the other, timeout on both will sooner or later occur and the deadlocking exchange will fail.
It is therefore correct to say that centralized control implies hierarchical decomposition of programs with monodirectional communication among levels and suffers from deadlock because the scheduler does not know the future status of programs, whereas decentralized control implies horizontal disaggregation of programs with bidirectional communications between grouped programs and suffers from racing because there is no central scheduler that knows the present status of programs.

3.5. Misconceptions about decentralized control

From the above point of view, it follows that most systems where message passing is central are conceptually badly justified, especially where it is claimed that such systems do provide sound foundations for distributed processing. The reason for which message passing may seem to be good for decentralized control is that, if schedulers do not cooperate, a global memory does not exist: it is thus not meaningful for a program to pass an object's id to another, as the other cannot use the id to refer to the object. It is thus necessary, under decentralized control, to pass the object themselves, and not their ids, as message passing seems to do. But under message passing the communication is mono directional, and this makes no sense unless there is some central control that acts as monitor of the communication; if there is not, it is not sensible to send messages to another program without knowing whether it still exists, and the only way to have some confidence about this is to request the other program to barter some object back as a kind of acknowledge.

Moreover, passing copies of objects (as message passing does) is troublesome, since there is always the stale copy problem. If a readers & writers scheme is used to solve it, this is much better with argument passing, that is more natural with centralized control and global memory.

Bartering, instead, conceptually implies that the bartered object is the original one (not a copy), even if in the actual implementation this may not be true (requiring some additional protocol among processes to get around the problem). The philosophy of decentralized control is that every object, even if it appears to have the same contents of another, is not a copy, because if there is no global memory it makes no sense to ask whether the ids of two objects are identical or not. LISPers will realize that this amounts to say that while centralized control is based on the EQ predicate, decentralized control is based on the EQUAL predicate, and that the problem of message passing is that it creates, in the visibility of programmer, objects that are EQUAL to some original, not being EQ, while the programmer wants them to be EQ (to synchronize under centralized control, implied by message passing).

3.5.1. Only bartering is logically sound

The rest of this paper is devoted to the problem of how to define and implement, with the current hardware, the communication primitives for distributed control. It is clear that the proposed implementation will be a fixup solution, because of the inadequacy of current language concepts and hardware structure.
The definition of the correct protocol for decentralized control follows from the above discussion; it

is required that when two processes on distinct machines with uncooperating schedulers want to communicate, the objects themselves be passed and not their ids, and that every object from one process shall be bartered against an object from the other. The barter shall either complete in a predefined time interval or be considered to have failed. Ids cannot be passed because there is no central control that makes a global memory environment possible, and communication must be bidirectional and subject to timeout because there is no central control that guarantees that a partner exists and that a failure of the partner will be reported.

3.5.2. A case for a special protocol

For special applications, under the constraints of given technology, it may well be that a protocol that does not implement neither the global memory for centralized control nor full bartering for decentralized control is to be studied and implemented. An example of such a protocol is discussed in [1], and described with a Pascal program. The intended application area is that of real time control systems for large machinery (such as marine motors or power generation stations), implemented as linked, cooperating microprocessors. The protocol described looks to the user like a message passing one, but it is really implemented with an (hidden) barter against an acknowledge, and with timeout. The programmer, in fact, uses what seem mono directional message passing primitives; the mono directionality is in effect apparent, as the module that implements the protocol requires a brief ACK message back from the target of the message. This ACK is subject to timeout.
All this is appropriate, as in control systems usually the different processors have distinct and non simmetrical tasks (for example controlling a meter and giving commands) that are naturally expressed by the programmer with one-way communication notations, whereas the system implements the (apparently) one way message as a barter of a message against an ACK. With full barter each object is somehow an ACK to the partner's object; thus having an explicit ACK is inefficient. This is really true when communication is by its nature simmetrical; when it is not, most often the programmer would explicity program for barter of information against a simple ACK. The system proposed in [1] relieves the programmer from writing down that (almost dummy) half of the barter. It is clear that, were communication is symmetrical, this would entail using two (apparently) one way messages, and this would be less efficient than a simple barter; but, in the intended range of application, communication is tipically asymmetrical, as when commands or measures are sent.

In the special protocol proposed, timeout on the hidden ACKs is essential because, even if usually the system is fully duplicated, and the knowledge that a link or a partner failed isn't usually used to reconfigure the system in a degraded mode, it is necessary to know that to avoid hanging, and to possibly switch over to the secondary system on detection of failure of a component of the system.

4. SIMULATING A CENTRALIZED SYSTEM

Assuming that there exists an implementation of bartering, there is the possibility of using it to implement the information exchange that makes cooperation of schedulers possible.

4.1. Let the processors cooperate

It is usually true that programming with decentralized control is annoying. It would be perfectly sensible to use it to implement a fake centralized system, by connecting via barters a communication among the schedulers of the different machines, so as to give the illusion of a single centralized scheduler, and to simulate a common global memory by mapping all the memory boxes in the same virtual address space.
The role of barters in such a system is to provide reliable links for communication. The schedulers will barter the status of processes with other machines, and a scheduler whose barter with another times out may assume that all the programs managed by that scheduler failed, and report the situation to those programs on its machine that had called others in the machine that no longer answers.

5. CONSIDERATIONS ON THE MODEL OF BARTERS

It has been shown that distributed control of computing entails two different problems; one is a non problem, in that it is possibile to simulate it inexistence with no worse efficiency than not having the simulation; the other is a semantic difference; for which new rules and attitudes are needed. The two different resulting models, while non reducible one

to the other, and thus primitive in their own right, have a strikingly dissimilar (dual) structure. The problems left open by this paper are those of formalizing the semantics of the decentralized control model, in a way similar to that of Prof. Dijkstra, and of defining and implementing suitable linguistic mechanisms to realize it. The rest of this paper tackles the solution to this second problem.

6. INTRODUCING THE IMPLEMENTATION OF BARTERING

It is now interesting to discuss how to provide the linguistic instruments (definition and implementation) to support this model. It is assumed that the scheduler of each machine in which bargainers (the programs that use barter) are executed shall provide the services to perform bartering. In order to allow for painless reconfiguration of a system when bartering is done by bargainers on the same machine is contemplated, with an efficiency hack. It shall also be possible to ask the scheduler to verify the outcome of a barter; this shall entail repeating the barter to add safety if necessary.

6.1. The efficiency of bartering

It shall be obvious from the following description that bartering is inherently more efficient than sending-receiving, in the case of decentralized control.

6.2. The "barter" primitive

The essential operation requested by bargainers (the processes that barter objects) may be described as

```
FUNCTION barter
    (VAR obj: an_obj; with: an_edge)
    : boolean;
```

Whose meaning may be taken to be: in the variable obj there is an object; barter it with whoever is at the other end of the edge so that the variable obj will now contain what was given back in barter. The edge is a simple data structure to describe to the barter module the (logical) connection between the two bargainers. For the barter to succeed, the edge data structure shall have been established by both bargainers using the same unique id, and the exchange of objects shall complete before the timeout period.

7. THE EDGE DESCRIPTOR

The edge descriptor is not really essential to the conceptual model of bartering; it is just useful to record the characteristcs of the logical connection between the bargainers. For example, it will hold the value of the timeout period (that is assumed to be constant for any barter on the same edge in this paper), the safety factor (explained later on), and the mailbox used to deposit objects being exchanged. The fact that the data structure describing the connection exists does not imply that the connection exists; the connection may cease to exist on any barter.

7.1. Using edges simplifies conventions

The edge data structure is also useful to put together the two bargainers. There is only one item of information that a bargainer need to know about its possible partner: its name. This point is very important: in no way the two bargainers can communicate if they don't know each other's name, and this name cannot therefore be bartered. The name must be known a priori from both processes. In order to allow the programmer to easily select such names, it is proposed not to name bargainers but the conceptual edge connecting them; in this way a bargainer needs to know only the name of the edge, to communicate with whoever is at the other end.

7.2. The fields of an edge descriptor

There is obviously a restriction on edge names: they must be unique on a given machine. The scheduler's barter support component shall therefore keep a directory of such edges; this shall also be necessary to deliver objects arriving from other machines to the mailbox of the correct edge. It is assumed in this paper that there exist (physical) links among machines, and that the communication along (logical) edges is multiplexed on the appropriate link. It is assumed for the sake of simplicity that if an edge between two bargainers exists, the link on which it is multiplexed is direct between the two machines. This may not be the (real) case; it is assumed that someone else will trouble to provide the (virtual) link, via a physical connection or a logical channel.

7.3. Establishing an edge

When a bargainer wants to barter with another bargainer, it shall use the procedure:

```
PROCEDURE init_edge
  (VAR edge: ref_edge;
   its_name: name_edge; its_link: ref_link;
   its_safety: a_natural; its_timeout: a_period);
```

This procedure will (possibly) build an edge descriptor to describe an edge through the given link, with the given name, and so on. The declaration of an edge descriptor will be something like:

```
an_edge =
    RECORD
        name: edge_name;
        link: ref_link;
        next: ref_edge;
        box: a_box;
        timeout: a_period
        safety: a_natural;
        proposed: a_natural;
    END;
```

Where the name is the name of edge, link the link through which it passes, next is the pointer to the next edge descriptor on the list of edges passing through the link, box is the mailbox that makes barter possibile, timeout is the timeout period for waiting for the other bargainer's object, safety is the number of safety checks to perform with this edge, and proposed is the safety factor proposed by the other bargainer (the one used will be the maximum of the two).
When an edge descriptor is initialized, the most complicated job is to put the edge descriptor in the directory of edges associated to that link.
When a barter is requested, the actions performed are those of putting one's object in the mailbox, and waiting for the other bargainer to put its, and then picking it up. The exact algorithm is somewhat more complicated than the most straightforward one because the box may hold only one object, but simulates a device that can hold two objects, and because a slightly simplified treatment is given to barter between bargainers on the same machine.

7.4. The role of the box in the edge

The box holds only one object at a time, but is used to simulate a device that can hold two objects. This device is conceptually analoguos to the rotating dish often found in banks, railways stations, convents. Such a dish is the only means of communication between a customer and a clerk. It is divided in two halves by some barrier (often of glass). The customer puts the money on its half of the dish, the clerk the ticket on his own half, and when both halves are full, the dish is rotated, the clerk takes the money and the customer the ticket.
In order to make this simulation, access to the box is serialized; the first task deposits its object, and waits for the other bargainer to come, find the box full, fetch the object in it, and deposit its own; this allows restart of the first that fetches the object and goes on. In more detail, a box contains two status indicators, "first" and "second" that indicate in which imaginary half of the dish the (single) object in the box is.

The box may be declared as follows:

```
a_box =
    RECORD
        obj: an_obj;
        half: RECORD first, second: a_fact END;
    END;
```

The "fact"s are half.first and half.second, and they indicate which half is currently full.

8. BARTER WITH A BARGAINER ON THE SAME MACHINE

In order to perform a barter, there are two possibilities, local or remote barter. The two possibilities are simmetrical; local barter is the easiest to describe (see fig. 1).
This procedure is very simple. First it checks via 'test and when' whether the first half is full; if so, it initiates the barter, putting the first half to full and waiting for the partner to fill the second half. If it is already full, the objects are exchanged and the first half is thus emptied and the second filled. This restarts the other bargainer that picks up the answer and thus empties the second half too.

In this protocol there are some things specific to local bartering; first, the barter always succeeds (there is no doubt as to whether the partner is able to answer - it is on the same machine); then there is no need of timeout and safety (for the same reason).

9. BARTERING WITH A REMOTE PARTNER

Bartering with a remote partner should be similar; the problem is to simulate the presence of both bargainers on the same machine. This is simple to do. A box is needed at both ends of an edge.

```
FUNCTION local_barter
    (VAR obj: an_obj; edge: ref_edge)
    : boolean;

VAR other: an_obj;

    PROCEDURE initiate_barter;
        BEGIN (*initiate barter*)
                (*deposit our object in the box*)
            (*when(box.half.first,false;*)
            box.obj := obj;
            now(box.half.first,true);
                (*and fetch the answer*)
            when(box.half.second,true);
            obj := box.obj;
            now(box.half.second,false);
        END (*initiate barter*);

    PROCEDURE complete_barter;
        BEGIN (*complete barter*)
                (*fetch in temporary location and deposit*)
            when(box.half.first,true);
            when(box.half.second,false);
            other := box.obj; box.obj := obj; obj := other;
            now(box.half.first,false);
            now(box.half.second,true);
        END (*complete barter*);

BEGIN WITH edge^DO BEGIN (*local barter*)
    IF test_and_when(box.half.first,false)
        THEN initiate_barter
        ELSE complete_barter;
    local_barter:= true;
END END (*local_barter*);
```

Fig. 1

Almost everything goes on as in the local case, but when a bargainer should deposit its object in the box, instead transmits it to the other machine where its agent will get it and deposit it in the box accessible to the partner. In this way, the agent represents the remote bargainer, and the situation is almost identical.

Two additional considerations are that when waiting for the agent of the other bargainer to deposit ots object, the bargainer shall timeout after a certain period, and that in case of successful barter some safety repetitions of it may be done to check that the barter was really completely successful.

9.1. The new components of the program

This design calls for four new components of the system: the first is the remote barter procedure, that invokes the sender and waits for the partner's agent; the second is the sender, that envelopes an object with its destination edge/box names and transmits it to the other machine; the third is on the other machine and is the receiver, that gets an envelope and passes the object in it together with its destination edge/box names to an agent; and the fourth is the agent that finds the edge/box to which an object is directed and delivers there the object to the partner of the remote barter.

9.2. The program to do remote barter

The remote bartering function may be described as in fig. 2.

9.3. Consideration on remote bartering

The remote bartering module is somehow longer than for local bartering; this is due to the existence of the two possible situations in which the barter on one side may begin (whether the partner's obj has already arrived or not), that are dealt with in slightly different ways for efficiency (in one case transmit then wait on mailbox for fetch - in the other fetch immediately then transmit), and to the option of repeating the barter some times for safety (to check that everything went well - comparing the retransmitted obj with the old copy).
A safety factor may be specified, that is a number of times the basic barter must be repeated for added checking. If bartering were done only once, it would be impossible for each partner to check that the bartered object arrived to the other correctly.

```
FUNCTION one_barter
    (this       : an_obj;
     VAR that   : an_obj;
     edge       : an_edge)
    : boolean;

VAR bartered: boolean;

    PROCEDURE initiate_barter;
        BEGIN (*initiate barter*)
                (*transmit the object along the edge to our agent*)
            (*when(box.half.first,false;*)
            sender(this,edge);
            now(box.half.first,true);
                (*and fetch the other obj when,if it arrives*)
            bartered := timed when(box.half.second,true,timeout);
            that := box.obj;
            now(box.half.second,false);
        END (*initiate barter*);

    PROCEDURE complete_barter;
        BEGIN (*complete barter*)
                (*fetch the already present obj and transmit ours*)
            when(box.half.first,true);
            when(box.half.second,false);
            that := box.obj; sender(this,edge);
            now(box.half.first,false);
            now(box.half.second,true);
        END (*complete barter*);

BEGIN WITH edge^DO BEGIN (*one barter*)
    IF test and when(box.half.first,false)
        THEN initiate_barter
        ELSE complete_barter;
    one_barter := bartered;
END END (*one barter*);
```

Fig. 2

```
FUNCTION repeat_barter
    (this: an_obj; VAR that: an_obj;
     edge: ref_edge; times : a_natural) : boolean;

VAR bartered : boolean; previous : an_obj;

BEGIN WITH edge^DO BEGIN (*repeat barter*)
    bartered := true;
    WHILE bartered AND (times GT 0) DO
        BEGIN
            times := times-1; previous := that;
            bartered := one_barter(this,that,edge);
            bartered := bartered AND (previous = that);
        END;
    repeat_barter := bartered;
END END (*repeat barter*);

FUNCTION remote_barter
    (VAR obj : an_obj; edge : ref_edge) : boolean;
VAR bartered : boolean; own : an_obj;

BEGIN WITH edge^DO BEGIN (*remote barter*)
    own := obj;
    IF one_barter(own,obj,edge)
        THEN bartered :=
            repeat_barter(own,obj,edge,max(safety,proposed))
        ELSE bartered := false;
END END (*remote barter*);
```

Fig. 2 (cont.)

To ensure the objects arrived without being corrupted, a repetition of the barter enables to check that the object sent to the partner arrived (because one of the sends must occur between two receives, and one may assume that if one receives an object the one sent before has arrived) and that the one received was that one (because it is retransmitted by the partner). In fact bartering twice may be quite satisfactory.
It may also appear that retransmission of the whole object for checking is excessive; this scheme gives good safety (it may however occur that the partner receives a wrong copy of the object, and on retransmitting it the exactly opposite error occurs), but it may well be convenient to retransmit only some unique id or some CRC or similar of the object.

It may be argued that the bartering module should deal only with simple bartering and leave any additional checks of any kind to the higher level protocols. This is certainly a correct idea, and is the reason for which no sequence checking (to check for the case an object is completely lost) is enforced by the bartering module given here.

9.4. The agent for remote bargainers

The bartering system described here needs also an agent, that is that module that listens messages sent to it through a given link and, on behalf of bargainers at the other end of the link, delivers the object in the message to the bargainer's partner, thus performing a deposit in the mailbox at the end of the edge in the machine in which the agent resides.

```
PROCEDURE agent
    (link : ref_link);

VAR packet : a_packet; received : boolean;

    FUNCTION get_packet
        (VAR stream : a_stream; VAR packet : a_packet;
         interval : a_period) : boolean;
    BEGIN WITH stream DO BEGIN (*get packet*)
        get_packet := timed get stream(stream,packet,interval);
    END END; (*get packet*)

    PROCEDURE deliver
        (packet : a_packet; list : queue_edge);

    VAR edge : ref_edge;

        PROCEDURE find_edge
            (VAR edge : ref_edge; name : name_edge;
             VAR queue : queue_edge);
        VAR next : ref_edge; stopit : boolean;
        BEGIN WITH queue DO BEGIN (*find edge*)
            edge := first.edge; stopit := false;
            REPEAT
                IF edge = NIL
                    THEN stopit := true
                    ELSE stopit := name = edge^name;
                IF NOT stopit THEN
                    BEGIN next := edge^next; edge := next; END;
            UNTIL stopit;
        END END; (*find edge*)
    BEGIN WITH list DO BEGIN (*deliver*)
        find_edge(edge,packet.dest,list,look);
        IF edge <> NIL THEN
            BEGIN WITH edge ^ box DO BEGIN
                proposed := packet.proposed;
                IF test and when(half.first,false)
                    THEN deposit(packet.obj,box,half.first)
                    ELSE deposit(packet.obj,box,half.second);
            END END;
    END END; (*deliver*)

BEGIN WITH link^DO BEGIN (*agent*)
    WHILE partner = remote DO
        BEGIN received := get_packet(inp,packet,interval);
            IF received THEN deliver(packet,edges);
        END;
END END; (*agent*)
```

Fig. 3

The agent's job is very simple; the only two noteworthy parts are polling the link and searching for the right edge descriptor.

9.4.1. Peculiarities of the agent

Polling is necessary simply because Pascal has no interrupt handling facility; the search for the correct edge descriptor is performed having all of them linked in a simple list to the link descriptor, and by performing a simple sequential scan of the list. This allows for dynamic establishment and deletion of edge descriptor; it should also be almost efficient when one has few edges through a link.

9.4.2. The link descriptor type declaration

The data structure describing a link is very simple: a link shall contain two stream descriptors, one for sending and the other for receiving messages (bartering implies bidirectionality); the beginning of the list of descriptors of edges through that link; the polling interval. The latter two items are not present if the link is a fictious link for local bartering, in which case it simply works as a catalog for the local edges.

```
a_link =
                    RECORD
        EDGES : list_edge;
        CASE other_side : (local, remote) OF
            remote :
                (inp, out : a_stream;
                 interval : a_period)
    END;
```

9.4.3. The agent program

The agent module itself is shown in fig. 3; the really interesting part is the deliverer.

10. CONCLUSIONS

As it may be easily seen, implementing bartering with conventional primitives is very easy, efficient, and safe. In effect bartering is obviously more efficient than an analogous scheme based on one way communication, because one way communication implies acknowledging overheads that are useless to bartering. Moreover, bartering is conceptually sounder then one way communication, and implies some synchronization not possible with the former. Also operating system overheads should be much lower for a barter than for send-receive (simply because the number of expensive user to system to user environment changes is minor).

10.1. Barter would be better with new hardware

It must be emphasized that implementing barter with low level send-receive, apart from being distasteful, is not really correct. In fact some troubles would be avoided if a medium physically more suitable to bartering could be used. Having one would not be certainly difficult, as bartering is already used by the technical people. For example the carrier detect mechanism in a modem is the epitome of the kind of mechanism a true implementation of barter would use to check for continued existence of the edge, and to be able to distinguish between link failure and partner failure.

10.2. Bartering is already implemented somewhere

In fact what this implementation of bartering attains, for example in the timeout detection mechanism (that is the equivalent, mutatis mutandis, of carrier detect) is the simulation of modems or many aspects of telephone systems .... This means that in fact other people, by trial and error, has since long discovered the necessity of untrusting barter. The idea that bartering was necessary came to the author when he saw the revolving dish of a railway station counter. Many other examples of the logic of bartering may be found, both in devices and in protocols (for example certain legal-financial conventions).
Implementations of bartering should be used to communicate across networks. In some way this has been already done. For example, in many networks with local routing algorithms, barter is used to exchange information about which nodes are working. In effect the streams over which bartering is performed may well be implemented as network virtual channels or the like; so that bartering may occur at two different levels, at the low level of the network software, for communication between physically adiacent nodes, and at the higher level of inter process communication between logically adiacent processors. In fact one of the most intelligent uses of a good implementation of bartering is to let the operating systems of different machines to exchange information so as to give programs on all of them the illusion of a single operating system.

11. REFERENCES

[1] V. Bacchini, Tesi di laurea, Università di Milano, Istituto di Cibernetica, A.A. 1979-8O

[2] Lauer, Needham, ON THE DUALITY OF OPERATING SYSTEMS STRUCTURES, Operating Systems Review, April 1979

[3] PROCEEDINGS OF ACM INTERPROCESS COMMUNICATION SYMPOSIUM, Operating Systems Review, July 1975

[4] Needham, Walker, THE CAMBRIDGE CAP COMPUTER, Proc. 6th ACM Symposium on Operating Systems

[5] APOLLO DOMAIN ARCHITECTURE, Apollo Computer Inc., February 1981

[6] Bensoussan et al., THE MULTICS VIRTUAL MEMORY, Comm. ACM, may 1972

[7] Baker, ACTOR SYSTEMS FOR REAL TIME COMPUTATIONS, MIT Tech. Report 197, Laboratory of Computer Science

[8] Larington et al., THE MU5 MULTICOMPUTER COMMUNICATION SYSTEM, Trans. on Computer, IEEE, genuary 1977

[9] Lampson, Redell, EXPERIENCE WITH PROCESSES AND MONITORS IN MESA, Comm. ACM, february 1980

NOTE SU UNA TECNICA DI DOCUMENTAZIONE DI PROGETTO (°)

E. Bertolotti (*)

Riassunto

Viene presentata la tecnica usata nella stesura della documentazione di un progetto (hardware e software) riguardante la realizzazione di un registratore di stripe magnetica.
L'obiettivo è stato quello di produrre delle specifiche la cui consultazione fosse agevolata da una struttura gerarchica con diversi livelli di dettaglio, senza alterare lo standard di documentazione già in uso, e peraltro lungamente collaudato, presso la HISI.

Abstract

We describe the method used in the production of the documentation for the project (hardware and software) of a stripe reader-writer.
The goal of this method is to produce documents easy to consult, organized in hierarchichal fashion with several levels of complexity, without changing the HISI documentation standards.

1. INTRODUZIONE

Scopo della presente nota è di illustrare il metodo con cui è stato affrontato il problema della produzione di documentazione durante il progetto di un registratore di stripe magnetica svolto presso la Direzione Tecnica Progetti Speciali della HISI.

Una stripe magnetica è un supporto di dimensioni standard usato per la registrazione in applicazioni quali badge, carte di credito, tessere a scalare, ecc. Nel caso specifico, si tratta di stripe poste sui libretti bancari. Il registratore in oggetto è realizzato con un Processor 8085.

Il tipo di progetto per il quale il metodo di documentazione descritto nel seguito è stato sperimentato riguarda, quindi, un piccolo sistema che ha richiesto la realizzazione, oltre che di un package firmware di governo del device, anche di un hardware intelligente in grado di gestire i segnali fisici provenienti dalle testine magnetiche e l'interfacciamento mediante linea seriale con l'utente.

2. STESURA DELLA DOCUMENTAZIONE

Il metodo è essenzialmente basato sull'applicazione di un approccio top-down nella stesura della documentazione, in accordo con gli standard di documentazione HISI, a partire dalle PFS (Product Functional Description). A tale scopo le PFS, contenenti la specifica dei requisiti del prodotto finale, sono state scritte ponendo l'accento sui seguenti aspetti:

a) *sinteticità del documento*
È stato compiuto lo sforzo di contenere il paragrafo standard "Functional Description of the Product" in poche pagine di linguaggio naturale nelle quali, pur se solo accennate e da specificarsi nell'ulteriore documentazione prodotta, vengono evidenziate *tutte* le esigenze alle quali il prodotto deve rispondere.

b) *Organizzazione delle PFS in brevi periodi con senso compiuto*, che specifichino le esigenze alla base del progetto espresse in termini di funzionalità e prestazioni del prodotto. Tali periodi sono stati strutturati in modo da poter essere poi agevolmente esplosi negli altri documenti previsti dallo standard HISI (EPS e PDD, vedi più oltre) fino ad ottenere un livello di documentazione contenente tutti i dettagli necessari.

c) *Definizioni dei termini sistemistici* raggruppate esclusivamente in un glossario allegato al documento per non invalidare la sinteticità dello stesso.

Dalle PFS così ottenute si sono quindi ricavati, *mediante esplosione di ogni singolo periodo*, i vari documenti di EPS (Engineering Product Specifications), che costituiscono le specifiche funzionali e di architettura dei moduli Hardware e Firmware.

I periodi componenti le PFS costituiscono i "Tito-

(*) Honeywell Information Systems Italia, Direzione Tecnica Progetti Speciali, Borgolombardo.

(°) La tecnica di documentazione descritta nella presente nota è legata allo sviluppo della metodologia PSPN ed è stata suggerita da G. Degli Antoni. Tale tecnica è stata utilizzata anche in "Una descrizione informale degli aspetti caratterizzanti Unix, un sistema operativo strutturato" di M. Casazza, in Note di Software 13/14.

li'' dei paragrafi di cui si compongono le EPS; il contenuto di ogni paragrafo è quindi, sostanzialmente, il raffinamento del rispettivo titolo. Naturalmente, ciò è risultato possibile, e particolarmente agevole, grazie al modo con cui sono state concepite le PFS.

La stessa metodologia è stata applicata per la stesura delle PDD (Product Design Description), il documento che definisce l'architettura di dettaglio del prodotto. Le PDD sono state infatti ricavate, seguendo lo stesso criterio, come ulteriore raffinamento delle EPS, periodo per periodo.

In particolare, in questo nuovo passo, a differenza del precedente, è sorta la necessità di dettagliare, oltre che frasi in linguaggio naturale, anche frasi in linguaggio formale, come ad esempio brani di specifica in Pascal di moduli firmware. In questo caso, le EPS contengono una descrizione in Pascal della struttura generale dei moduli; le PDD raffinano tale codifica, sempre in Pascal, fino a un livello di dettaglio sufficiente per la produione del codice assembler.

Naturalmente, nel caso di moduli già disponibili ed adeguatamente documentati, si è trattato semplicemente, a livello PDD, di richiamarne la bibliografia.

La Fig. 1 mostra uno schema dei possibili casi che si sono presentati nella procedura di raffinamento della documentazione.

3. ALCUNI ESEMPI

Diamo, in fig. 2 e 3, alcuni esempi tratti dalla documentazione

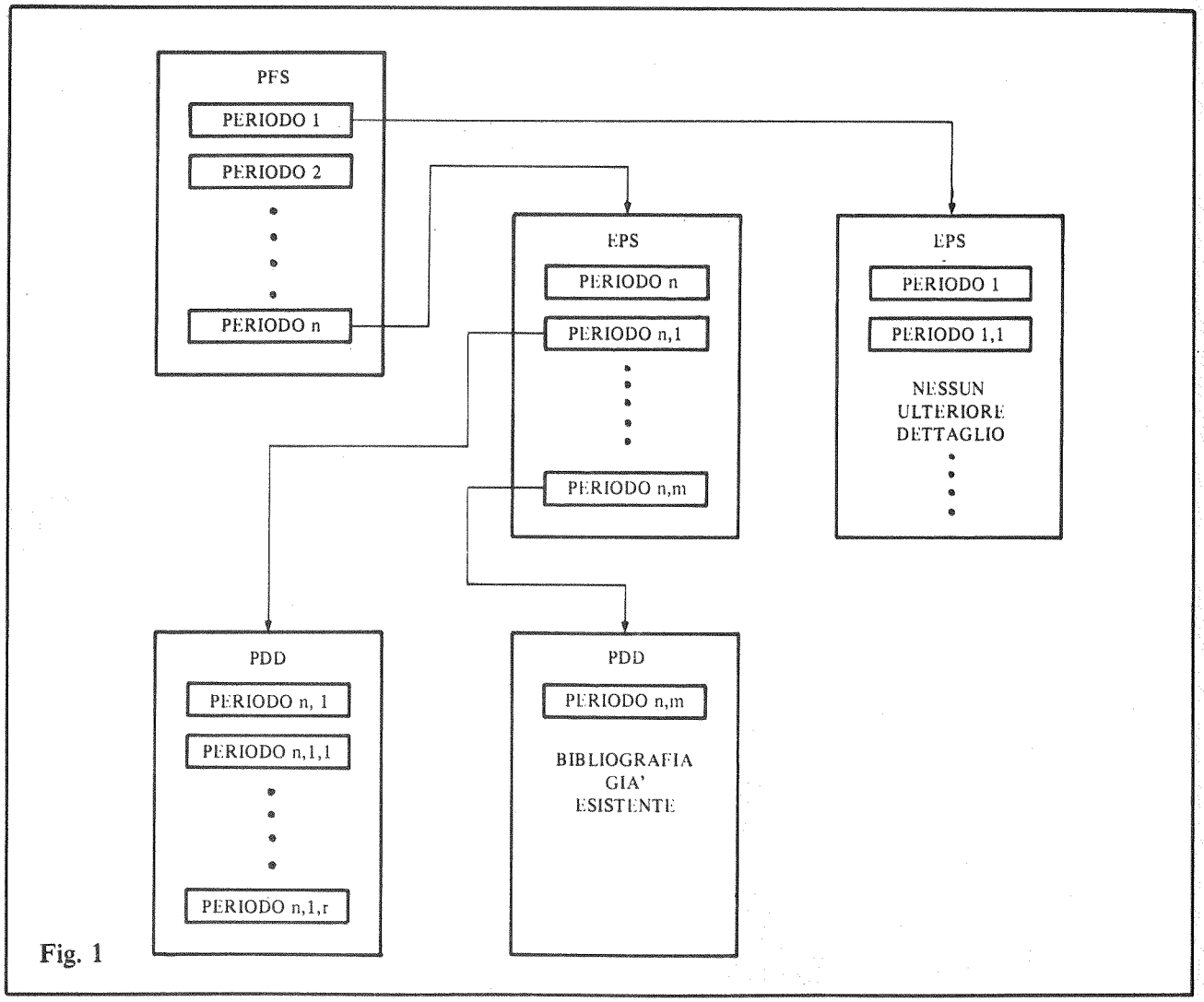


Fig. 1

da PFS "STRIPE READER WRITER"

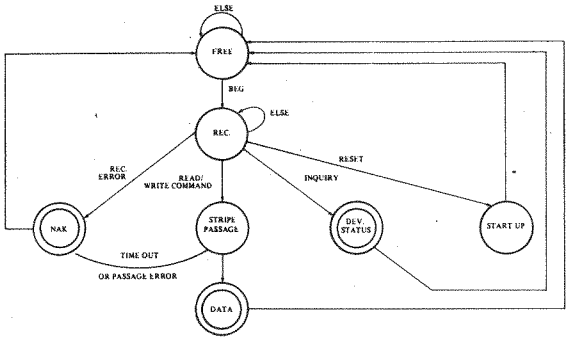
The I/O to the user serial line is implemented with a simplified protocol that receives functional commands and transmits data and device status.

da EPS "STRIPE READER WRITER"

The I/O the user serial line is implemented with a simplified protocol that receives funtional commands and transmits data and device status

«OMISSIS»

Commands received and data transmitted are shown in the following status diagram:



«OMISSIS»

The reception and transmission are handled by the line handler module.

- line handler module

```
program handli
begin (* called for receive or transmition handling *)
if < receive > then < command ACK. >
                    (* find command received *)
                else < transmission buffer building > ;
                    (* prepare transmission *)
end
```

«OMISSIS»

da PDD "STRIPE READER WRITER"

«OMISSIS»

- line handler module

```
program handli
begin (* called for receive or transmission handling *)
if < receive> then (* find command received *)
                    case < receive command > of
                    read command  : < signal stripe reading >
                    write command : < signal stripe writing >
                    inquiry       : < ack device status >
                    devicereset   : < device start up >
                    end
                else (* prepare transmission *)
                    case < transmition buffer > of
                    stripe data   : < prepare stripe buffer >
                    device status : < prepare device status >
                    notack        : < prepare nack reply >
                    end
end
```

«OMISSIS»

Fig. 2 - Esempio 1

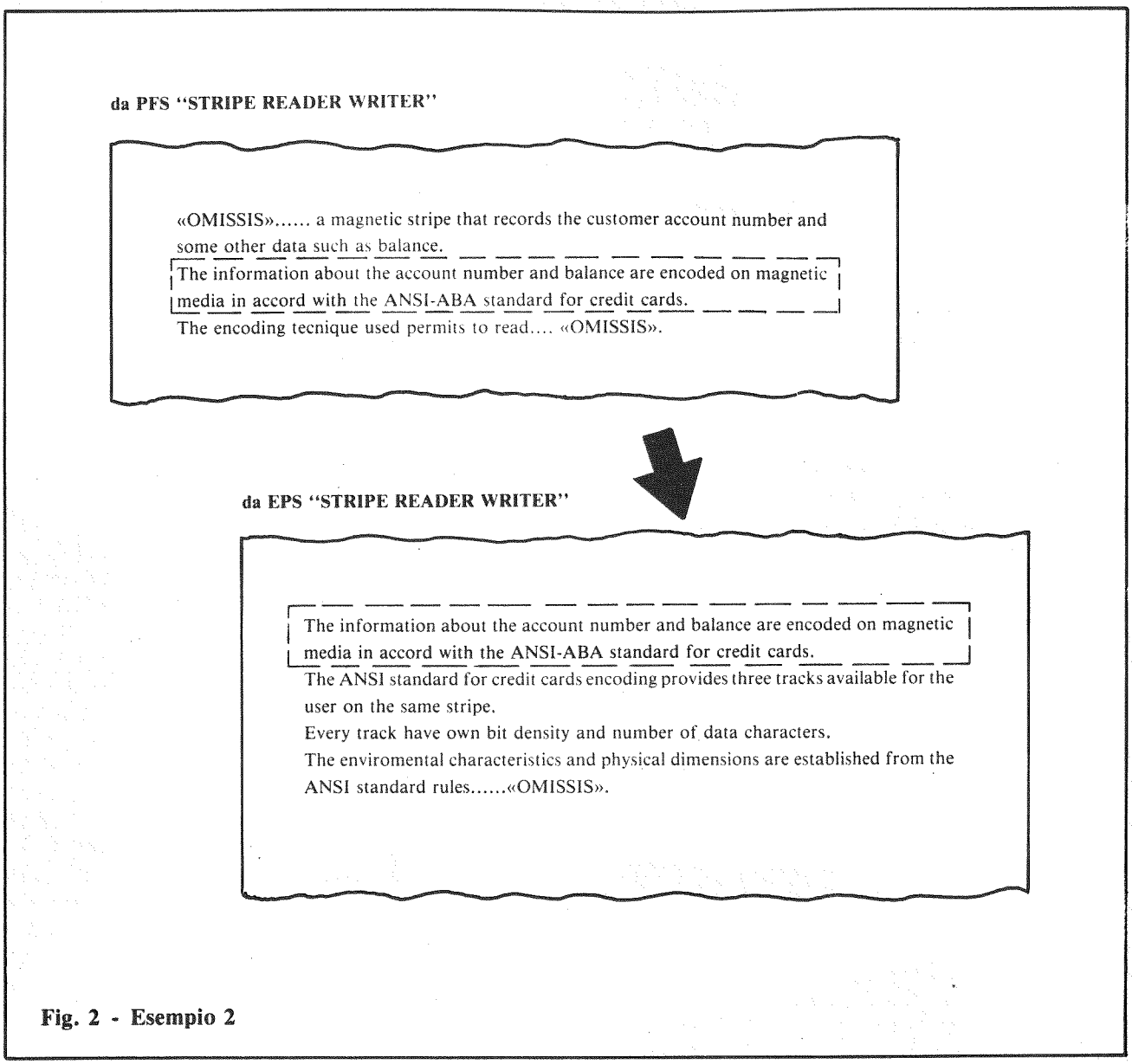


Fig. 2 - Esempio 2

4. CONCLUSIONI

I principali vantaggi riscontrati nel metodo descritto stanno nell'aver prodotto una documentazione con le seguenti caratteristiche:

a) *Struttura gerarchica consultabile a livelli*
Le specifiche di diverso livello di dettaglio sono tra loro connesse da una struttura gerarchica che agevola l'accesso e la ricerca di informazioni del grado di complessità desiderato.

b) *Possibilità di cross-referencing fra i vari documenti:*
Risulta cosí particolarmente agevole la rilevazione di eventuali lacune nella documentazione o incongruenze tra una fase e l'altra dello sviluppo del progetto.

Una difficoltà riscontrata nell'applicazione del metodo sta, invece, nella sua sovrapposizione a standard di documentazione preesistenti a cui si è dovuto adeguare. Un'ulteriore difficoltà ha presentato il tentativo di minimizzare la quantità di informazioni aggiuntive da integrare nelle specifiche una volta completato il loro raffinamento; ciò ha prodotto la necessità di ripetute revisioni del documento iniziale (PFS) per l'aggiunta di informazioni mancanti, evidenziate nelle fasi di raffinamento della documentazione.

IL SEGNALIBRO

In questa rubrica vengono segnalati libri, articoli o studi di recente pubblicazione che, a giudizio della redazione o dei lettori di NOTE DI SOFTWARE, rivestono un particolare interesse nell'ambito dei temi a cui questa rivista è dedicata.
Le citazioni fra virgolette, se non ne è indicata la fonte, sono tratte dai lavori stessi.

Ada

• Barnes, J.G.P., AN OVERVIEW OF ADA, Software - Practice and Experience, vol. 10, n. 11 (novembre 1980), 851-887

"This paper commences with an outline description of the development of the Ada programming language and its position in the overall language scene. The body of the paper is an informal description of the main features of the final language as revised after the Test and Evaluation phase of the DoD project. Comparison with the preliminary version is made where appropriate."

• PROCEEDINGS OF THE ACM-SIGPLAN SYMPOSIUM ON THE ADA PROGRAMMING LANGUAGE, Boston, Massachusetts, December 9-11, 1980, in SIGPLAN Notices, vol. 15, n. 11 (novembre 1980)

"[...] These papers were selected from almost one hundred papers submitted less than one year after the first, wide distribution of the Ada Reference Manual. These papers are an impressive demonstration of the diverse activity that has been stimulated by Ada. Some of the most interesting and important current results are reported here. [...]"

Indice: USE: J.B. Goodenough, The Ada Compiler Validation Capability; D.S. Notkin, An Experience with Parallelism in Ada; R.E. Fairley, Ada Debugging and Testing Support Environments; A.G. Duncan, J.S. Hutchison, Using Ada for Embedded Microprocessor Applications; COMPILERS: G. Goos, G. Winterstein, Towards a Compiler Front-End for Ada; G. Persch, G. Winterstein, M. Dausmann, S. Drossopoulou, Overloading in Preliminary Ada; P.A. Belmont, Type Resolution in Ada: An Implementation Report; M.S. Sherman, A Flexible Semantic Analyzer for Ada; J. Rosenberg, D.A. Lamb, A. Hisgen, M. Sherman, The Charrette Ada Compiler; A. Hisgen, D.A. Lamb, J. Rosenberg, M. Sherman, A Runtime Representation for Ada Variables and Types; M. Sherman, A. Hisgen, D.A. Lamb, J. Rosenberg, An Ada Code Generator for VAX 11/780 with Unix; B.M. Brosgol, TCOL-Ada and the "Middle-End" of the PQCC Ada Compiler; STYLE: D.C. Luckham, W. Polak, A Practical Method of Documenting and Verifying Ada Programs with Packages; W.D. Young, D.I. Good, Generics and Verification in Ada; B. Krieg-Brückner, D.C. Luckham, ANNA: Towards a Language for Annotating Ada Programs; L.A. Clarke, J.C. Wileden, A.L. Wolf, Nesting in Ada Programs Is for the Birds; TASKING: L. MacLaren, Evolving Toward Ada in Real-Time Systems; W. Eventoff, D. Harvey, R.J. Price, The Rendezvous and Monitor Concepts: Is There an Efficiency Difference?; D.R. Stevenson, Algorithms for Translating Ada Multitasking; TRANSITION: G.L. Filipski, D.R. Moore, J.E. Newton, Ada as a Software Transition Tool; P.F. Albrecht, P.E. Garrison, S.L. Graham, R.H. Hyerle, P.Ip, B. Krieg-Brückner, Source-to-Source Translation: Ada to Pascal and Pascal to Ada; R.B.K. Dewar, G.A. Fisher Jr., E. Schonberg, R. Froehlich, S. Bryant, C.F. Goss, M. Burke, The NYU Ada Translator and Interpreter; EXECUTION MODELS AND ARCHITECTURE: F.C. Belz, E.K. Blum, D. Heimbigner, A Multi-Processing Implementation-Oriented Formal Definition of Ada in SEMANOL; H.H. Lovengreen, D. Bjorner, On a Formal Model of the Tasking Concept in Ada; L.J. Groves, W.J. Rogers, The Design of a Virtual Machine for Ada; J.M. Bishop, Effective Machine Descriptors for Ada.

• Wegner, P., PROGRAMMING WITH ADA - AN INTRODUCTION BY MEANS OF GRADUATED EXAMPLES, Prentice-Hall, Inc., Software Series, 1980, pp.xi + 211

"The purpose of this book is to present an introduction to Ada for programmers with at least one year's experience in a higher-level language such as Fortran. The degree to which a language is understood and accepted depends not only on its technical quality but also on the quality of expository materials. The aim of this book is to provide a readable and relatively painless introduction to the language quite early in its development so that the

process of acceptance can be accelerated. The method of presentation - by means of examples which vary from relatively trivial illustrations of programming language principles to nontrivial developments of "real" programs - has not, to the author's knowledge, been tried before on this scale. The author has for a number of years wanted to write a programming language introduction along these lines, and Ada has provided a worthy excuse. The book is incomplete in a number of respects. It is a book about a language which is still a moving target. Id does not consider certain 'advanced' features of the language such as representation specifications. It is not intended to replace the reference manual but merely to supplement it. The author plans to develop in the summer of 1980 an updated version of this book which reflects the language changes to be introduced as a result of language evaluation. However, it is felt that meanwhile the present volume may serve a useful purpose for the many programmers who are interested in Ada now.

Indice: 1. An Overview of Ada, 2. Basic Language Features, 3. Data Description, 4. Modularity and Program Structure, 5. Multitasking.

Pascal

● Shimasaki, M., Fukaya, S., Ikeda, K., Kiyono, T., AN ANALYSIS OF PASCAL PROGRAMS IN COMPILER WRITING, Software - Practice and Experience, vol. 10, n. 2 (febbraio 1980), 149-157

"A method and results of static and dynamic analysis of Pascal programs are described. In order to investigate characteristics of large systems programs developed by the stepwise refinement programming approach and written in Pascal, several Pascal compilers written in Pascal were analysed from both static and dynamic points of view. As a main conclusion, procedures play an important role in the stepwise refinement approach and implementors of a compiler and designers of high level language machines for Pascal-like languages should pay careful attention to this point. The set data structure is one of the characteristics of the Pascal language and statistics of set operations are also described."

● Moffat, D.V., A CATEGORIZED PASCAL BIBLIOGRAPHY (JUNE, 1980), SIGPLAN Notices, vol. 15, n. 10 (ottobre 1980), 63-75

"This bibliography includes references to most of the material about Pascal written in English. The references are mainly to texts, monographs, and articles in professional journals. I made no attempt to include articles from trade journals such as Datamation, or from popular magazines such as BYTE and Creative Computing; that task I leave for a rainy day."

● Faiman, R.N., Kortesoja, A.A., AN OPTIMIZING PASCAL COMPILER, IEEE Transactions on Software Engineering, vol. SE-6, n. 6 (novembre 1980), 512-518

"The architecture of a production optimizing compiler for Pascal is described, and the structure of the optimizer is detailed. The compiler performs both interprocedural and global optimizations, in addition to optimization of basic blocks. We have found that a high-level structured language such as Pascal provides unique opportunities for effective optimization, but that standard optimization tecniques must be extended to take advantage of these opportunities. These issues are considered in our discussion of the optimization algorithms we have developed and the sequence in which we apply them".

● Foster, V.S., PERFORMANCE MEASUREMENT OF A PASCAL COMPILER, SIGPLAN Notices, vol. 15, n. 6 (giugno 1980), 34-38

"The data obtained above is strictly valid only for the particular Pascal compiler (Pascal 6000-3.4 release 2) operating on a particular computer (CDC Cyber 172) operating under a particular operating system (NOS/BE). It clearly identifies portions of this compiler that should be studied by anyone interested in decreasing compilation time. The performance measurement also produced results that should be kept in mind by any compiler designer interested in execution efficiency [...]"

Testing

● Glass, R.L., REAL-TIME: THE "LOST WORLD" OF SOFTWARE DEBUGGING AND TESTING, Communications of the ACM, vol. 23, n. 5 (maggio 1980), 264-271

"Real-time debug and test is still a 'lost world' compared to the 'civilization' developed in other areas of software, says Robert L. Glass. From a survey of current practice across several projects and companies, he defines a state of the art of this problem area and suggests improvements which will ease the practitioner's task."

● Celentano, A., Crespi Reghizzi, S., Della Vigna, P., Ghezzi, C., COMPILER TESTING USING A SENTENCE GENERATOR, Software -Practice and Experience, vol. 10, n. 11 (novembre 1980), 897-918

"A system for assisting in the testing phase of compilers is described. The definition of the language to be compiled drives an automatic sentence generator. The language is described by an extended BNF grammar which can be augmented by actions to ensure contextual congruence, e.g. between definition and use of identifiers. For deep control of the structure of the produced sample the grammar can be described by step-wise refinements: the generator is iteratively applied to each level of refinement, producing at last compilable, complete programs. The implementation is described and some experimental results are reported concerning PLZ, MINIPL and some other languages."

Tools

● Triance, J.M., Yow, J.F.S., EXPERIENCES WITH a SCHEMATIC LOGIC PREPROCESSOR, Software - Practice and Experience, vol. 10, n. 10 (ottobre 1980), 791-800

"A COBOL processor to implement Michael Jackson's program design language, Schematic Logic, has been written and used in the Computation Department at UMIST. This paper investigates the problems inherent in using preprocessors of this type and details the advantages which accrue from the preprocessor when teaching and using the Jackson Method."

● Vaucher, J.G., PRETTY-PRINTING OF TREES, Software - Practice and Experience, vol. 10, n. 7 (luglio 1980), 553-561

"A pleasing layout of printed tree structures is difficult to achieve automatically. The paper points out three main problems. First, the horizontal position of a node on the printed page depends on global consideration of the position of other nodes; secondly, the physical characteristics of printers require scanning the tree in left-to-right top-to-bottom sequence; finally, page overflow for wide trees must be handled. These problems are illustrated by analysing the shortcomings of a simple printing algorithm. A suitable general binary tree printing algorithm is presented and its adaptation to other types of trees is shown."

● Oppen, D.C., PRETTYPRINTING, ACM Transactions on Programming Languages and Systems, vol. 2, n. 4 (ottobre1980), 465-483

"An algorithm for prettyprinting is given. For an input stream of length n and an output device with linewidth m, the algorithm requires time $O(n)$ and space $O(m)$. The algorithm is described in terms of two parallel processes: the first scans the input stream to determine the space required to print logical blocks of tokens; the second uses this information to decide where to break lines of text; the two processes communicate by means of a buffer of size $O(m)$. The algorithm does not wait for the entire stream to be input, but begins printing as soon as it has received a full line of input. The algorithm is easily implemented."

● Gimpel, J.F., COUNTOUR - A METHOD FOR PREPARING STRUCTURED FLOWCHARTS, SIGPLAN Notices, vol. 15, n. 10 (ottobre 1980), 35-41

"Countour is a program whose purpose is to graphically illustrate a program's structure. It operates by bounding the scope of loops and conditionals by solid (or nearly solid) lines. When compound statements are embedded in other compound statements, one obtains, rather than confusion, a rather pleasant display reminiscent of the contour lines of a topographical map. Aside from its visual appeal, the method has the advantage that it makes far fewer demands on the reader's linguistic expertise and so may be used for presenting algorithms in an almost language-independent manner (a kind of structured flowchart)."

Metodologie di disegno

● Teorey, T.J., Fry, J.P., THE LOGICAL RECORD ACCESS APPROACH TO DATABASE DESIGN, ACM Computing Surveys, vol. 12, n. 2 (giugno 1980), 179-211

"Database management systems have evolved to the point of general acceptance and wide application; however a major problem still facing the user is the effective utilization of these systems. Important to achieving effective database usability and responsiveness is the design of the database. This paper presents a practical stepwise database design methodology that derives a DBMS-processable database structure from a set of user information and processing requirements. Although the methodology emphasizes the logical design step, the activities

of requirements analysis and physical design are also addressed. The methodology is illustrated with a detailed example. Performance trade-offs among multiple users of a single integrated database are considered, and the relationship between short-term design and design for flexibility to changing requirements is discussed. Many steps in the database design process can be assisted with proper use of computer modeling techniques and other tools, such as requirements analysis software. The example design problem and its solution steps serve to point out when and where current technology can be effectively used."

● Cowan, D.D., Graham, J.W., Welch, J.W. Lucena, C.J.P., A DATA-DIRECTED APPROACH TO PROGRAM CONSTRUCTION, Software - Practice and Experience, vol. 10, n. 5 (maggio 1980), 355-372

"The present paper discusses a method of program construction based on the specification of the data types. The input and output data types and the mapping between them are specified at a high level of abstraction and this non-procedural specification is used to develop a program schema. The data type and mapping specifications are modified to include a concrete representation of the data and these are used to expand the program schema into a program. A graphical representation for data and program specifications is also introduced and it is shown how this can simplify the techniques and be very useful in program construction. The method is illustrated by developing two programs -the line justifier program described by Gries and the bubblesort. [...] This approach resembles the approach used by Jackson."

Un libro sulla programmazione di sistema (in Pascal Plus)

● Welsh, J., McKeag M., STRUCTURED SYSTEM PROGRAMMING, Prentice-Hall International, Series in Computer Science, 1980, pp.xii + 324

"The purpose of this book is to demonstrate the application of structured programming to the construction of system programs - in particular compilers (which are typical of many similar text-handling programs) and operating systems (which are typical of many real-time systems).
The book begins by summarizing (in Section 1) the structured programming style and notations to be used. Section 2 and 3 then present the development of a complete compiler and a complete operating system, with working code for each, in a suitable high-level language. The language used is an extended version of the programming language Pascal, known as Pascal Plus. This extended language has been implemented at the Queen's University of Belfast on two different computers, and the programs presented in this book have been compiled and run there. A portable Pascal Plus system is now under preparation. However a reader should be able to implement these programs in any suitable language on his own computer, thereby obtaining some practical experience, as well as theoretical understanding, of structured system programming.
The book should be useful to three classes of reader.
1. For those learning structured programming the book presents two complete case studies of its application to larger programs, showing clearly the particular problems of size which large programs present and how they may be tackled.
2. For those studying compilers or operating systems the book provides a corresponding case study of the implementation of established compiler or operating system techniques within the clear logical framework of a structured program.
3. For professional programmers already engaged in system programming the book demonstrates how structured programming techniques can be applied successfully in their current area of work, and may encourage them to adopt these techniques if they have not already done so."

Reti di Petri

● W. Brauer, editor, NET THEORY AND APPLICATIONS - PROCEEDINGS OF THE ADVANCED COURSE ON GENERAL NET THEORY OF PROCESSES AND SYSTEMS, Amburgo, 8-19 ottobre 1979, Lecture Notes in Computer Science, Springer-Verlag, 1980, pp.xiii + 537

Si tratta del materiale presentato al corso organizzato dal Dipartimento di Informatica dell'Università di Amburgo in cooperazione con il GMD di Bonn, di cui si è scritto nel numero 10/11 di Note di Software (rubrica Avvenimenti), che riporta l'elenco di tutti i contributi qui riprodotti.

Decision tables

● Sethi, I.K., Chatterjee, CONVERSION OF DECISION TABLES TO EFFICIENT SEQUEN-

TIAL TESTING PROCEDURES, Communications of the ACM, vol. 23, n. 5 (maggio 1980), 279-285

"Sequential testing procedures for checking the rule-applicability of the decision tables encountered in practice are usually found to be minimum-path-length trees. On the basis of this observation, an algorithm is developed for converting decision tables to efficient decision trees. A criterion is defined for estimating the minimum expected cost of the tree in terms of rule probabilities and condition-testing costs and is utilized in arriving at the efficient decision tree. The algorithm is applicable to general limited-entry decision tables and can also bee employed for manual coding."

● Maes, R., AN ALGORITHMIC APPROACH TO THE CONVERSION OF DECISION GRID CHARTS INTO COMPRESSED DECISION TABLES, Communications of the ACM, vol. 23, n. 5 (maggio 1980), 286-293

"The decision grid chart is investigated as an intermediate form in constructing decision tables; a very natural extension to the basic format is proposed. Five alternative procedures for the conversion of decision grid charts into compressed decision tables are given. All but one avoid the storage of the fully expanded decision table in memory."

Articoli vari

● Patel, A., Purser, M., SYSTEMS PROGRAMMING FOR DATA COMMUNICATIONS ON MINICOMPUTERS, Software - Practice and Experience, vol. 10, n. 4 (aprile 1980), 283-305

"This paper examines some of the special considerations which apply to the development of such software for minicomputers. These are treated under the following general categories:
— loading and throughput
— the external interfaces of the system software
— the internal structure of the system software
— developing the software"

● Chambers, J.A., Sprecher, J.W., COMPUTER ASSISTED INSTRUCTION: CURRENT TRENDS AND CRITICAL ISSUES, Communications of the ACM, vol. 23, n. 6 (giugno 1980), 332-342

"The use of computers to assist in the learning situation in a simulation, game, tutorial, or drill and practice mode is reviewed on an international basis with centers of activity identified in the United States, Canada, the United Kingdom, and Japan. The use of the computer as an adjunct to support learning is compared to its use in a substitution mode. Evaluative studies of CAI are reviewed and costs are examined. The critical issues of CAI are enumerated and analyzed as they pertain to computer hardware, CAI languages, and courseware development and use. The future of CAI is briefly sketched from the viewpoints of individuals prominent in the field. Finally, conclusions are drawn and recommendations are offered to help ensure the most educationally cost-effective use of CAI in learning situations."

● Ledgard, H.F., A HUMAN ENGINEERED VARIANT OF BNF, SIGPLAN Notice, vol. 15, n. 10 (ottobre 1980), 57-62

"There is no question that BNF, or one of its myriad variants, is popularly used to define programming languages. Yet almost every time a language is defined, yet another variant is used. I believe that the reason for this proliferation is the continuing dissatisfaction with the details of this notation. I offer here a very simple, human engineered variant of BNF. In no way is this variant a technical contribution, but rather several small ideas for improving *human* comprehension."

● Brinch Hansen, P., Fellows, J.,THE TRIO OPERATING SYSTEM, Software - Practice and Experience, vol. 15, n. 11 (novembre 1980), 943-948

"This paper is an overview of the Trio system which enables users to simultaneously develop and execute programs at three terminals. The system consists of an operating system written in Concurrent Pascal and a set of standard programs written in Sequential Pascal. The system has been used on a PDP 11/55 minicomputer since spring 1979. This work concludes 5 years of experience with the first abstract language for concurrent programming."

Libri sulla programmazione

● Conway, R., Archer, J., Conway, R., PROGAMMING FOR POETS - A GENTLE INTRODUCTION USING PASCAL, Winthrop Publishers, Inc., 1980, pp.xiv + 330

"This is a book about programming, for readers who don't expect to do much programming themselves, but who would still like to understand what it is all about.

Our objective is to explain what programs are like, without having to require that the reader become proficient in their construction. Our technique is primarily to supervise the reading of selected programs, rather than the writing of programs. Programs are, of course, written in a programming language, and reading them requires a certain minimal literacy in such a language. To achieve this literacy, we present a gentle introduction to the basic concepts and constructs of programming, and expect the reader to become comfortable with these ideas by writing some simple programs. Up to this point, our approach is quite conventional. It becomes radical when we abruptly switch to presenting programs that are considerably longer and more complicated than the reader could be expected to construct on his own. We discuss these programs, not to help the reader become able to write similar programs, but to help him understand what they are, how they work, and something about why they are that way.

After this brief course in program literature, we undertake a discussion of such topics as the difference between programmers and users, and the difference between programming languages and applications programs. We discuss the nature of "computer errors" and the responsibility for them. Finally we try to explore the limits of programming by discussing processes such as "understanding" and "creating."

Our objective is to achieve an understanding of the programming process comparable to what a student of programming per se might achieve after two or three courses. But we attempt to do this in a single course, since this is probably all that most students are willing to allocate to this area as a culturally broadening experience. The only possible way to achieve this acceleration is to carefully avoid trying to make programmers out of these students."

Indice: Part I: ELEMENTS OF PROGRAMMING: 1. What is a Program?; 2. Data-Printing Programs; 3. Statement Repetition; 4. Conditional Statements; 5. Output Format and Titling; 6. Computing New Values for Variables; 7. Variables with Multiple Values; 8. Program Testing; 9. Preliminary Examples; Part II: PROGRAM EXAMPLES: 1. Some Text-Editing Programs; 2. A Concordance Program; 3. A Statistical System; 4. Translation of Natural Languages; 5. Matching and Selection; 6. Random Processes; 7. Interactive Computing Systems; Part III: THE NATURE AND LIMITS OF PROGRAMMING: 1. Users and Programmers; 2. Some Limits on Programming; 3. Errors in the Computing Process; 4. Abuses of the Computer; 5. The Intelligence of the Computer; 6.Popular Wisdom About the Computer; APPENDICES: A. Summary of the Poet's Subset of Pascal; B. Additional Topics in Pascal; C. String Processing in Standard Pascal.

Honeywell

Honeywell Information Systems Italia

Via Laboratori Olivetti - Pregnana Milanese

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA
Via Viotti, 5 - Milano